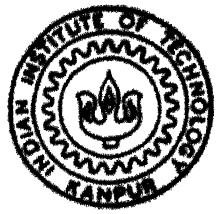
OPTIMIZATION OF FORTRAN 90 PROGRAMS : ALIAS ANALYSIS

by

B. MURALI



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

March 1994

Optimization of Fortran 90 programs : Alias analysis

A thesis submitted
in partial fulfilment of the requirements
for the degree of

Master of Technology

by

B. Murali

to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

March, 1994

CSE-1994-M-MUR-OPT

17.3.94

CERTIFICATE

It is certified that the work contained in the thesis titled *Optimization of Fortran 90 programs : Alias analysis*, by **B. Murali** has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

March, 1994

Dr. Sanjeev Kumar Aggarwal
Assistant Professor,
Department of Computer Science
and Engineering,
IIT Kanpur

To the stars in their courses that had guided me throughout my stay here at IITK.

ACKNOWLEDGEMENTS

Before coming to IITK, I used to consider data flow analysis and, indeed, optimization in compilers, a mystery beyond comprehension. Hence, I was very much interested in working in this area in order to demystify it. Consequently, when Dr. Aggarwal offered this project, I jumped to it. Over the last one year, I have learnt quite a lot in this field and have, needless to say, managed to demystify it. Thanks to this end goes primarily to Dr. Sanjeev Kumar Aggarwal for introducing me to this field, for being a constant source of guidance and encouragement and for his patience with me throughout my thesis work.

One person who has been a constant source of help and company during my stay at IITK is "Nassimi" (*i.e.*, Narasimhan). I thank him for both, especially his help with the various tricky problems I had faced during my thesis work and for the long hours (nights on end) of "discussions" we used to hold on worldly issues.

Thanks also goes to the "all knowning" Sai Baba (as Sai Rama hates being called) for all his knowledgeable tips on everything under the sun from OS to compilers, C language to data bases, and so on.

My stay at IITK, particularly at Hall V, was made lively by understanding friends which include Partha, YSS, Shyam, GCP and others. In particular, GCP had made my stay here lots more enjoyable by essentially introducing me to SF and CR, and by providing the lighter moments in IITK. Thanks goes to all these guys.

Thanks also goes to my parents and sister who have been much patient with me during those long periods when I did not correspond with them. They have been a source of constant encouragement to me.

Finally, the astro-club and Mme. Karnick and her French class also contributed in making my stay here extremely enjoyable and educative. I shall forever remember both.

B. Murali

Abstract

The work outlined in this report relates to the optimization phase of the vectorizing compiler for the Fortran 90 programming language, which is an ongoing project at IIT, Kanpur.

The first aspect of this work is the analysis of the aliasing patterns in the input program, including the effects of procedure calls by reference and the effects of COMMON and EQUIVALENCE statements.

The next aspect of this work concerns the computation of reaching definitions information for the given program. This involves interprocedural analysis as well as the determination of ambiguous and unambiguous definitions and uses in the program.

Finally, the alias information and reaching definitions information have been employed in performing constant propagation and folding in the given program. In this project, we have worked at the source level of the programs, preserving their control structures as we go about the process of optimization. Among other things, this allows us to use the optimizer to enhance the vectorizability of the programs by uncovering more loops in the programs having constant initial, final and increment values of the loop variable. The vectorizer can then be run on the output of the optimizer and so extract more parallelism from the code.

Contents

List of Figures

Chapter 1

Introduction

The vast majority of scientific software, developed till date, have been written in Fortran. The recent advances in computer architecture has seen the emergence of vector processors as a powerful means to achieve higher speed in such processing as may require a large number of vector operations. It is therefore convenient to have scientific software mould itself to the utilization of languages offering vector manipulating statements. Fortran 90 (or, in its pre-standard form, Fortran 8x) offers some constructs for specifying vector operations at the source level ([For8x]). Considering that the collection of and investment in existing software in scientific work is enormous, and most of them are scalar code, it would be worthwhile to design a *vectorizing* compiler – one that takes scalar code and generates equivalent vector code from it. The ongoing Fortran 90 compiler project at the Indian Institute of Technology, Kanpur is just such an attempt, targeting the Fortran 77 – Fortran 90 class of languages as the input language.

1.1 Structure of the compiler

The structure of the compiler is depicted in figure 1.1. The different phases of the compiler and their functions, in brief, are as follows :

- **Preprocessor :** This phase handles the compiler directives such as file inclusion, and presents to the lexical analyzer the expanded version of the program, so that it has a uniform picture of the program being compiled. This stage takes the given Fortran 90

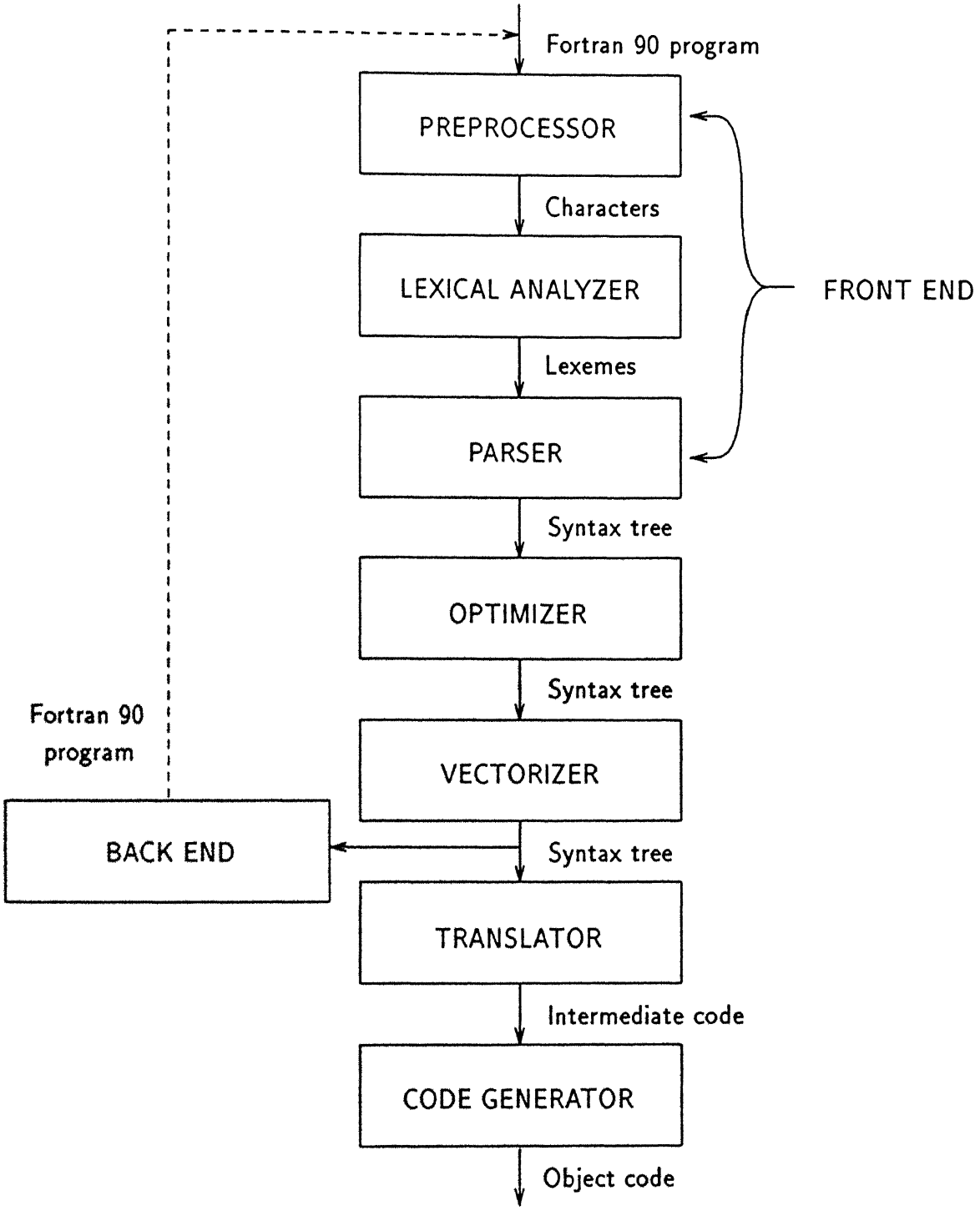


Figure 1.1: Structure of the Fortran 90 compiler.

program as input and produces the characters of this expanded version of the program as output.

- **Lexical Analyzer :** This phase takes as input the stream of characters it receives from the preprocessor and it outputs the lexemes formed by these (according to the definition of the Fortran 90 language).

- **Parser :** This phase uses the lexemes it receives from the previous phase and recognizes the syntactic structures formed by these lexemes, as defined by Fortran 90. This stage forms a tree representation of the given program wherein each node represents an operator or a language construct, such as an *if-then* statement, while its children represent either its operands (if this node represents an operator) or other language constructs that go into forming the construct represented by this node. Such a representation of the program is called an *abstract syntax tree* or simply syntax tree.

- **Optimizer :** This phase takes the syntax tree representation of the given program, generated by the parser, and performs code-improving transformations on it. What this means is that this phase will change some definitions of variables, move code around in the syntax tree, and perform a number of similar transformation on this syntax tree such that the code corresponding to this transformed tree will execute more efficiently than the original one or will take less space than the original one, or both. The output of this phase is also a syntax tree, albeit (possibly) a different one from the one given as input. This phase is optional in that it need not be used in every run of the compiler, *e.g.*, when the user is developing the program he/she may not want to use the optimizer since the intention at this stage would be to check the correctness of the program rather than to make it efficient. Under control of flags, given at the command line during invocation of this compiler, the user can force the compiler to bypass this stage.

- **Vectorizer :** This phase takes the syntax tree representation of the given program, generated by the optimizer, and converts DO-loops involving array element assignments to equivalent vector assignment instructions[1] wherever possible. The output of

[1] Actually it converts subtrees representing DO-loops into subtrees representing equivalent vector instructions.

this phase is also a syntax tree, and this phase is also optional for the same reasons as the the optimizer phase.

- **Back End :** This phase takes the syntax tree output of the vectorizer, and generates an equivalent Fortran 90 program for this syntax tree, a process called *unparsing*. This phase is an aid to inform the user of the transformations performed on the given program, such as by the optimizer and/or the vectorizer. This phase is also optional, as the user may need to use it only during program development, and not when compiling the final program. The dashed line in figure 1.1, from the back end to the parser, indicates that the output of the back end can be fed back to the compiler after the user has made changes to it, since this output is a valid Fortran 90 program. The user may make such changes to this generated program as may be necessary to aid the compiler in optimizing and/or vectorizing the given program. Fortran 90 provides a set of compiler directives aimed at forcing the compiler to do certain optimizations and/or vectorizations which it may not do otherwise. This may happen because the compiler lacks the necessary information to do these transformations and conservatism demands that under such circumstances the compiler not perform these changes. However, the user may be in a better position to judge the safety of such transformations and so may be able to guide the compiler in performing them.

- **Translator :** The syntax tree obtained from the vectorizer is converted by this phase into a forest of trees, wherein each internal node represents some operator and the leaf nodes represent the operands. This is done by translating all array references, structure references, function calls, *etc.*, into low level operations. The output of this phase is a sequence of preorder traversals of these trees.

- **Code Generator :** The main functions of this phase is to allocate registers for the various operands mentioned in the output of the translator, as well as to select the machine instructions to be used for the various operations mentioned therein. The output of this stage is the object code for the target machine.

1.2 The work done here

The work outlined in this report concerns itself with the code optimization phase. Specifically, we shall be concerned with the following three aspects of code optimization :

1. Two variables in the same part of a program are considered to be *aliases* of each other if it is possible for both of them to refer to the same memory location at the same point of time. We shall be concentrating on the algorithms used to determine such alias pairs within the Fortran 90 programs, the utilization of this aliasing information in the subsequent analysis and transformations on the given program, as well as the issues related to the implementation of these algorithms. These have has been discussed at length in Chapters 3 and 4.

2. A definition of a variable (*i.e.*, a statement defining this variable) is said to reach a particular use of this variable (*i.e.*, a statement using this variable), if within the program, control can flow from this definition to this use, without any other definition of the same variable coming in between[2]. In this report we shall also be looking at the application of the results of the alias analysis, mentioned in the preceding paragraph, in determining such reaching definitions information.

3. If the use of a variable is reached only by a definition of the same variable which assigns a constant value to it, then we can replace all references to this variable, in this use, by the corresponding constant value. This is called *constant propagation*. If, in this manner, all operands of a particular expression are found to have constant values, then this expression can be evaluated at compile time. This process is called *constant folding*. In Chapter 4 we shall be dealing with the use of the reaching definitions information to perform constant propagation and folding on the given Fortran 90 program. In fact, this optimization is most useful as an aid to the vectorization process as has been mentioned in the following Section.

1.2.1 The representation of the program

In this work the representation of the input Fortran 90 programs that has been used during the optimization phase is the syntax tree representation. By manipulating the syntax tree

[2]This is not strictly true, as we shall see in Chapter 4.

of the program we are able to deal with the program essentially at the source level. We have applied code analysis and transformations on this high level representation of the program in such a way as to maintain the language structure as we proceed through the various stages of the optimization process. This representation captures the syntactic structures of the input program and allows us to construct a Fortran 90 program from it using the back end. Following are the major reasons that motivated this choice :

- Optimization using the syntax tree representation of the input program preserves its control structures as far as possible. During the code generation phase, this will allow us to use the full power of advanced processors that provide sophisticated control structures at the hardware instruction level.

- The vectorizer, which follows the optimizer, assumes its input to be a syntax tree of the input program. By performing some code transformations such as the constant propagation and folding (mentioned previously), prior to the vectorization process, we can potentially increase the amount of vectorizable code. For example, the vectorizer considers only those DO-loops as candidates for vectorization which have constant initial, final and increment values of the loop variable. By performing constant propagation and folding, we may uncover more loops with constant values for these three parameters, which would have been otherwise considered to be variable and consequently excluded from the process of vectorization.

- By keeping the program still at the syntax tree level, we can use the unparser to generate a Fortran 90 program from it. This can be used to provide the user with a feedback as to the transformations done by the compiler on the program as well as those transformations which it has been prevented from doing. Ideally, the compiler should be able to report to the user the reasons for its failure in performing certain transformations. The user can then change his/her program, if possible, with an eye to aiding the compiler in optimizing and/or vectorizing it.

1.3 Organization of this report

This report has been organized into five chapters. The information covered in these chapters is briefly outlined as follows :

Chapter 1 outlines the following :

- Structure of the compiler.
- The work done here.

Chapter 2 deals with data flow analysis in general and discusses the following :

- Data flow analysis, in general.
- Some new paradigms for data flow analysis.
- Efficient constant propagation algorithms.
- Issues in aliasing, in brief.

Chapter 3 deals with alias analysis. This includes :

- Issues and algorithms for the analysis of aliases due to COMMON and EQUIVALENCE statements.
- Issues and algorithms for the analysis of aliases due to parameter passing by reference.

Chapter 4 includes the following issues :

- The building of the control flow graph.
- Computing uses and definitions due to non-call statements.
- Computing uses and definitions due to procedure calls.
- Interprocedural analysis.
- Reaching definitions analysis.
- Constant propagation and folding.

Chapter 5 concludes this report with the following :

- Integrating the optimizer with the rest of the compiler.
- Testing the optimizer.
- Summing up the work done so far.
- Future additions to the compiler.

Chapters 3 and 4 are the implementation related chapters. Therefore, they contain brief notes on the implementation of the various algorithms and the problems faced therein.

Chapter 2

Data flow analysis techniques

In this chapter we present, at a glance, a number of important advances in the field of *data flow analysis*, which is the process of collecting information about the way variables are used in a program. [ASU86] gives us a comprehensive survey of the techniques used in data flow analysis and code improving transformations. Here we shall examine some of the newer ideas that are different from the *classical* algorithms mentioned therein. We shall also touch upon the issue of aliasing in programs.

2.1 Traditional methods of data flow analysis

As said previously, [ASU86] is a collection of the different traditional methods of data flow analysis. We shall briefly look into these methods.

2.1.1 The control flow graph

In analyzing the given program, it is necessary to represent the control flow structure of the given program in a convenient manner. This is generally done by constructing a graph called a *control flow graph* [1] ([ASU86]) abbreviated to CFG. Nodes in this graph represent what are called *basic blocks*. A basic block is a sequence of consecutive statements in which flow of control, within the program, enters at the first statement of this sequence and leaves at the last one, without halt or possibility of branching except at the last statement. There is a directed edge from one node to another in this graph whenever control can flow, within

[1] Although [ASU86] prefers to call it simply a *flow graph*.

the program, directly from the last statement in the basic block corresponding to the first node to the first statement in the basic block corresponding to the second node.

2.1.2 Gathering information

The data flow information collected at various points in a program can be related among themselves using simple set equations. Most traditional data flow analysis schemes use the two set representation of data flow information used originally in [MoRe79]. In this scheme two sets are associated with each basic block in the CFG – one set called the IN set, at the input of the basic block, and another set called the **OUT** set, at the output of the block. These two sets form the data flow information that is being sought. The information at each node in the CFG is related to those at its predecessors and/or successors in it. Depending upon the type of such relation we have three types of data flow problems. These are :

- *Forward* data flow problems, in which the IN set of each node in the CFG depends on the OUT sets of each of its predecessors in it. The OUT set of a node is computed in terms of its IN set and the data definitions in the sequence of statements forming the basic block corresponding to this node. It is the IN set which is computed using data flow analysis and set equations relating this set with the OUT set of each of its predecessors.

- *Backward* data flow problems, in which the OUT set of each node depends on the IN sets of each of its successors in the CFG. The IN set of a node in the CFG is computed from its OUT set and the data definitions in this basic block. The OUT set of a node is expressed in terms of the IN sets of its successors using set equations and is computed using data flow analysis. Both this type of data flow problem and the previous one fall into the category of *unidirectional* data flow problems, since the information in these cases flows in only one direction – from a node to its successors or from a node to its predecessors, but not both ways.

- *Bidirectional* data flow problems, in which the IN set of each node depends on the OUT sets of its predecessors *and* the OUT set of each node depends on the IN sets of its successors in the CFG. Here, both these two sets have to be computed using data flow analysis and set equations to relate them to the appropriate sets of their predecessors or successors.

The relation between the information at one node and that at its predecessors/successors is expressed as *flow functions* which map the information at one node to that at another node, and are generally associated with the edges connecting these two nodes. These flow functions essentially constitute the set equations alluded to previously. When the information collected can take only one of two values it is represented in binary form *i.e.*, using 0 and 1). Then these flow functions become mappings from the set {0,1} onto itself. For example, the reaching definitions information can be stored at each node as the set of nodes whose definitions reach this node. This information has only two values – either a node is in this set or it isn't.

The standard method employed in computing the data flow information is to start with an initial approximation to the information at each node of the CFG. Then the flow functions are repeatedly applied on the information at the successors and/or predecessors of each node (as applicable) to the get the new value of the information at this node, until the information no longer changes at any node of the CFG.

The easiest means of implementing these sets is to use bit vectors. One bit is allocated for each possible item in the set and this bit is set or reset according as this element is in this set or not. The algorithms given in [ASU86] are mostly such bit vector oriented algorithms. The main attraction of these algorithms is the ability to use a single machine word to represent a set of as many elements as the number of bits in it, by assigning one bit in this word to each element in the set. The various set operations, such as union, intersection, *etc.*, can be implemented using single instructions such as bitwise OR, bitwise AND, *etc.*, performed on these words. This exploitation of the *word parallelism*, particularly where the word size is large, *e.g.*, 32 bits, is the main reason of choice of these algorithms.

2.2 New paradigms for data flow analysis

The major drawback of these traditional bit vector oriented algorithms is their time complexity. Since these algorithms keep iterating until the information at all nodes stabilize at some value, the upper limit on the number of iterations gives us the number of steps these algorithms will need. In each iteration these algorithms have to examine every node in the CFG and for each node they have to execute a bit vector step, involving the performance of the set operations using the bitwise operations provided by the machine. All three factors

- the number of iterations, the number of nodes visited in each iteration and the number of operations needed in a single bit vector step - can go up to the total number of nodes in the CFG. This can make the time complexity of such algorithms go up to $O(n^3)$ in the worst case, where n is the number of nodes in the CFG (*i.e.*, essentially the program size). The use of the word parallelism in the machine merely serves to reduce the constant factor in this complexity.

In order to improve the running time of this analysis phase, recent research has been diverted towards newer methods of data flow analysis that do not use the bit vectors used hitherto. Here we discuss the core ideas behind three such techniques, along with their relative merits and demerits. All these algorithms strive to reduce the execution time complexity of the data flow analysis process, in comparison to the traditional methods.

2.2.1 Static single assignment form of the CFG

In the traditional form of the control flow graph, the flow functions are not simple functions and the values generated by them need to be computationally determined at all nodes of the graph. A new form of the CFG has been suggested, which strives to reduce most of these functions to some simple form that would make the process of computing the values generated by them either simpler than before or eliminate it altogether, in some cases. This new form of the CFG satisfies the following properties :

1. Each programmer-specified use of a variable is reached exactly by *one* definition of the same.

2. The CFG contains some special functions, called ϕ functions, which are used to distinguish values of variables transmitted on distinct incoming control flow edges. Essentially, for each assignment to a variable v, we create a unique new name for this variable, say v_i, for the i^{th} assignment to v, and introduce a ϕ function, at each node in the CFG that has more than one predecessor. This ϕ function performs the assignment

$$v \leftarrow \phi(v_{i_1}, v_{i_2}, ..., v_{i_j}),$$

where j is the number of predecessor of this node in the CFG. This function will assign to v the value v_{i_k} whenever control reaches it along the k^{th} in-edge of this

node, assuming v_{i_k} to be that incarnation of v whose definition reaches this node along this in-edge. These ϕ functions must satisfy the following properties :

(a) If a CFG node Z is the first node common to two non-null paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$, that start at nodes X and Y containing assignments to v, then there should be a ϕ function for v at the entrance to Z.

(b) The use of each new name v_i for variable v is reached by exactly one assignment of v in the program.

(c) Along any control flow path, consider the use of the new name v_i for variable v (in the transformed program) and the corresponding use of v (in the original program). Then v and v_i have the same values.

3. Each variable has an assignment at the start node of the CFG, *i.e.*, the node which gets the initial control in the CFG.

This form of the CFG is called the *static single assignment (SSA)* form. The major advantages of this form of the CFG over the standard one are :

1. In the standard form of the CFG, data flow information might have to be recomputed each time branches are deleted during optimization. This may need yet another round of data flow analysis. In the SSA form of the CFG, it is easy to recompute such information since only one definition of a variable reaches any use of the same.

2. For each definition of a variable, it is often convenient to keep the list of all uses of this variable which are reached by this definition. Such a list forms what is called the *du-chain* of this definition ([ASU86]). These du-chains are more compact in the SSA form of the CFG than the standard one, since the number of du-chains in the SSA form of the CFG is at most equal to the number of edges in it.

3. As we shall see in Section 2.3, the SSA form is eminently suited for tracking redundant computations across control flow paths. This has been used in a number of algorithms on constant propagation in order to uncover more expressions as compile time constants.

4. Many of the flow functions, along the edges of this SSA form of the CFG, are either constants or identity functions, thereby eliminating the need to calculate the value generated by these functions.

More details on this form of the CFG has been given in [CFR89], which also gives an algorithm to construct it. This algorithm employs a concept called *dominance frontiers* in order to determine the points at which these ϕ functions have to be inserted.

The time taken by this algorithm (as modified in [CFR91]), is $O(n^2)$ (in the average case) and $O(n^3)$ (worst case), where n is the number of nodes in the CFG. This implies that algorithms that use the SSA form will inherit these corresponding bounds. This makes them asymptotically slower than the *slotwise analysis* algorithm of the next subsection. However, the pessimism in the worst case behaviour is much less here than in that method. But it must be remembered that converting a CFG to SSA form will increase the number of nodes, due to the introduction of the ϕ functions. So fewer the number of ϕ functions introduced by an SSA forming algorithm the more efficient it is. The algorithm given in [CFR89] constructs a *minimal* SSA form of a CFG, by keeping the number of ϕ functions introduced at a minimum.

2.2.2 Slotwise analysis

This is actually a modification of the traditional bit vector oriented algorithm mentioned in Section 2.1.2. In this algorithm, each bit position or *slot* is considered separately from the rest. For example, in computing the reaching definitions information, as said at the beginning of this chapter, the information sought is the set of definitions reaching any node. In traditional bit vector oriented algorithms, all definitions of all variables are considered together ([ASU86]). Slotwise analysis, on the other hand, considers each definition of each variable separately. At each node this definition either reaches it or not. Hence this information can be represented in binary form, using 0 and 1. By studying the ways of mapping the set {0,1} onto itself, the nodes having values to be determined computationally are separated from those having the default (trivial) values. In the case of the reaching definitions analysis, the nodes that are themselves definitions of the same variable, as the definition being considered in the current slot, will not allow[2] this definition to reach any of

[2]Under certain conditions, as we shall see in Chapter 4.

their successors in the CFG. Thus, in this slot, the flow functions along the edges to their successors shall always yield 0 irrespective of whether the information received at the input is 0 or 1 (where we assume that information value 1 at a point implies that this definition reaches that point, while information value 0 implies that it does not). By using a simple worklist based approach, this method confines the part of the CFG explored during data flow analysis to only those nodes that have non-default values for their bits. This method has been fully explained in [DRZ92].

The advantages of this algorithm are :

1. Slotwise analysis examines only a small fraction of the parts of the CFG that conventional algorithms would examine.

2. Standard algorithms need $O(n^2)$ bit vector steps (in the worst case) and $O(n)$ steps (average case), each step taking $O(n)$ time, where n is the number of nodes in the CFG. Hence, their overall worst case time complexity, as mentioned in Section 2.2 , can go up to $O(n^3)$. Here the time taken has been proved in [DRZ92] to be $O(n^2)$ in the worst case, which is an $O(n)$ improvement over the standard algorithm.

3. This algorithm can afford to enumerate all possible mappings forming the flow functions, since it examines only one slot at a time. Standard algorithms examine *all* slots together and so will need to enumerate all possible combinations of mappings from {0,1} to {0,1} for each of the slots. The number of such mapping combinations is exponential in the number of slots being considered, and it is not viable to enumerate all of them.

4. Some data flow analysis problems, such as available expressions ([ASU86]), are particularly amenable to slotwise analysis. This is because in such cases

   - the information in any slot can be computed, used and discarded before proceeding to the others, and
   - the information in most basic blocks in large programs will have default values. In computing available expressions, for example, one can note that most expressions will not be available in large parts of the program, their availability being confined to a few nodes only.

5. Since this algorithm considers a slot and discards information about it before proceeding to the next one, we can detect the cases where optimization at one node may open up new vistas for optimization that did not exist in the original program. These new opportunities for optimizations are called *second order effects*. In traditional methods, we would require yet another level of data flow analysis in order to discover such opportunities, because each optimization, which is performed using these techniques, uses only the information available at the start of the optimization and not the updated information as the optimization is proceeding. In slotwise analysis, one can reintroduce slots, to be processed once again, if it is found that new opportunities for optimization have been introduced at the corresponding nodes.

6. Some bidirectional problems can be solved efficiently using slotwise analysis by converting them to unidirectional problems and applying a quick correction to the solutions to these to get solutions to the original bidirectional problems. It must be noted that bidirectional problems may take more time using the traditional approach because information at a node can flow back and forth along edges in the CFG, thus needing a large number of iterations for the results to stabilize. This has been treated at length in [DRZ92].

However, this algorithm is not without its disadvantages. Some of these are :

1. There are many algorithms like live variable analysis, constant folding, *etc.*, which are not amenable to slotwise analysis. This is because in these cases we *cannot* consider each slot in isolation from the rest, since the analysis of each slot requires information at other slots as well.

2. In order to solve bidirectional data flow analysis problems using the technique suggested in [DRZ92], we need to introduce redundant nodes in the CFG (by a process called *edge splitting* - [DRZ92]). This can lead to an increase in the size of the CFG with a consequent increase in the running time of the algorithm.

2.2.3 Incremental analysis

Although discussed in this section, incremental analysis does not represent an altogether new paradigm for data flow analysis. It is, however, a method of data flow analysis that

strives to update the data flow information even after some optimizations have been done, in such a way as to take less time than would a complete rerun of the algorithm to compute this information. This technique avoids restarting the computation at all nodes by using the knowledge of the analysis made previously and of the changes made to the program subsequent to this analysis.

Most incremental analysis techniques are *imprecise* in that the solution obtained may be inferior to what would have been obtained by a complete rerun of the data flow analysis algorithm. [MaRy90] gives us an algorithm which is *precise* according to the preceding definition.

Here, the CFG is decomposed into subgraphs such that within each subgraph every node is reachable from every other node by a directed path[3], in other words, its *strongly connected components (SCCs).* For maximum efficiency these SCCs must *maximal* in that no additional node can be included into any of these SCCs without destroying their basic characteristics. By collapsing each SCC to a node, and having edges from one of these new nodes to another whenever there is an edge from some node in the SCC corresponding to the first node to some node in the SCC corresponding to the second one, we can get the *condensed graph.*

To each SCC they use an internal solution which is unchanged if no changes have been made to the part of program represented by it. This constitutes the *restricted problem* and it captures the data flow within this component. The *representative problem* takes into account the local effects by and on external information. This is expressible solely in terms of the local information at different SCCs. These two solutions are combined and sent to each successor SCC (successor as determined by the condensed graph) to form the global solution, which is then stored explicitly only at the head and exit nodes of the SCCs.

Local reinitialization of the information within an SCC is needed whenever changes occur to the part of the program represented by it. The global problem changes at the headers and exits of all SCCs. We need to repeat the data flow iterations only for nodes in the SCCs which have been either modified or have been newly created. The changed information is then propagated to the successors of such modified/new SCCs or to the targets of added/deleted edges.

The complexity of this algorithm is a function of

[3]Remember, a CFG is a directed graph.

- the size of the largest SCC in the CFG,
- the size of the condensed graph,
- the maximum number of nodes in an SCC which are relevant to the analysis, and
- the maximum number of variables in the part of the program represented by an SCC.

The major advantages of this algorithm are :

1. The algorithm achieves incremental updation of the data flow information without having to restart the computation at all the nodes in the CFG. Basically, this is a technique to identify the affected portions of the program and confine data flow analysis to such places.
2. If the condensed graph and the largest region are much smaller than the flow graph, and edge addition does not frequently join distant regions, then this algorithm is more efficient than the standard algorithms for precise incremental iteration by factors exceeding $n^{1/2}$ (as claimed in [MaRy90]).

However, this algorithm is heavily dependent on the condensation of the CFG (into its SCCs) leading to balanced decompositions, in order to perform efficiently. This means that a lopsided SCC will make this algorithm do extra work to achieve the same thing as restarting the traditional algorithms over the whole graph.

2.3 Discovering more compile time constants

A simple algorithm for constant propagation is to find out the number of definitions of a variable reaching any use of the same. If there is only one reaching definition and if this assigns a constant value to the variable, then we shall use this constant value to replace this use of this variable. Whenever all variables in any expression have been replaced by constant values, we can evaluate this expression. This may give rise to more constant definitions. This process is continued until there are no more uses of constant definitions. However, this does not discover a large class of constants as has been suggested by Kildall ([ASU86]). Therefore more sophisticated algorithms have been suggested by others in this field. We shall look into four such algorithms, in brief.

2.3.1 Kildall's algorithm

One of the first improvements is due to Kildall ([ASU86]), who introduces the notion of associating a lattice with the each slot, *viz.*, each variable in the program. This lattice has three levels – the highest one being *top*, the next one *constant* and the lowest one *bottom*. This notion has been explained at length in [ASU86], where they have used the terms *undef*, *constant* and *nonconstant*, respectively in place of the ones given by Kildall. Briefly the significance of these values, at any point in the program, are :

- *top* : This value implies that as yet one cannot say whether the variable possessing this value has a constant value at this point or not.
- *constant* : This value implies that all possible values that the variable can have at this point are constants and have the same value, so far.
- *bottom* : This value implies that the value of the variable cannot be guaranteed to be a constant at this point.

It is obvious from the preceding points that the information stored at each node is a three valued one.

In Kildall's algorithm each node is examined and the lattice value for each slot is initialized according to the definitions of the lattice values given above. Then each node in the CFG is examined. Each slot at this node, which has a value other than *bottom*, is considered and the value for this slot is made the lowest value for that slot in any predecessor of this node in the CFG. If this may cause the value of some slot in some assignment statement to differ from its stored value, then all successors of this node are reexamined. The algorithm terminates if there are no more nodes to examine.

This algorithm is simple and takes the same amount of time as the simple algorithm does, *viz.*, $O(EV^2)$ time, where E is the number of edges of the CFG and V is the number of nodes in it. The major advantage of this algorithm is that it does not confine itself to single definitions reaching any use in order to try and use the value defined therein at this use. Whenever more than one definitions of a variable reach any of its uses, this algorithm tries to find if all of them are constant definitions and if so whether they all define the *same* constant value for this variable. If so, then it uses this value at this use.

2.3.2 Using the du-chain graph

Kildall's approach uses the CFG as the graph along which the information is propagated. An alternative approach is to use the *du-chain graph* for this purpose. The du-chains (as explained in Section 2.2.1) give us the set of uses reached by each definition. The du-chains, viewed as actual edges from the definitions to the uses they reach, form a graph, which is the du-chain graph. The algorithm propounded by Reif and Lewis ([WeZa85]) uses such a du-chain graph, with some modifications, lowering and propagated the lattice values in it, while going directly from each statement modifying a variable to its uses. This method has the advantage that this graph is expected to be sparser than the CFG and hence the execution time, which is proportional to the size of the graph being considered for propagating the constant information, will be reduced.

Both the preceding two techniques discover what have been termed by Kildall as *simple constants*, which are, essentially, constants obtained without trying to find out the direction branches will take. In other words, no attempt is made to evaluate the expressions that control conditional branches. The next algorithm tries to avoid branches which can be determined, at compile time, to be non-executable.

2.3.3 Constant propagation evaluating conditional branches

[WeZa85] suggests two algorithms in order to work around a naivity in the previous two algorithms. This is the case where conditionals may be evaluated at compile time and untraversable branches may be ignored. This can lead to lesser number of reaching definitions and so open up more potential constants. The first of these algorithms actually forms a part of the more powerful second one. Hence, we shall discuss only the latter.

This algorithm uses a modified version of the du-chain graph, called the *global value graph*. Nodes in the CFG that satisfy the following properties are classified as *birthpoints* :

1. Each definition of a variable v is a birthpoint.

2. If node n has two or more in-edges and there is a node m, which is a birthpoint of a variable v, such that there is a birthpoint free path from m to n through one of the in-edges to n but not through the others, then n is also a birthpoint of v. A path from m is said to be birthpoint free if it does not go through any birthpoints for the variable v.

By the above definition of the term, a birthpoint essentially gets placed in the CFG wherever different information for a variable enters that node from different in-edges.

To each node, which is the immediate predecessor of a birthpoint for a variable v, which is not a definition of v, this algorithm adds an identity assignment of the form v←v. Such an assignment is added at this birthpoint also. This is used to cut off definitions that reach along non-executable branches. By definition, it's not too difficult to see that only one birthpoint of a variable can reach any of its use. Thus the introduction of the birthpoints, coupled with the identity functions, give us nothing but a different representation of the CFG in SSA form.

The global value graph is constructed over the same set of nodes as the CFG. Edges are added from a birthpoint of a variable to each uses of this variable reachable from this birthpoint.

Following this we examine both this global value graph and the CFG to traverse only constant conditional branches. Only nodes already known to be executable have their outgoing du-chain edges considered. If the node has value *bottom* then both the true and the false branches leading out of this node (if it is a conditional) are considered as executable and will be examined subsequently. If it's marked *constant*, then we consider as executable only that branch which is bound to be taken due to this constant value. If it has a *top* value, then we shall continue with other nodes which have been marked as executable and which haven't been examined subsequently. The algorithm ends when there are no more nodes which have been marked as executable and not considered in the above process after that.

This algorithm takes care of two conditions that the previous two algorithms cannot handle :

1. It does not follow branches which it can find to be unexecutable.

2. Even if both a definition and a use of the same variable are marked as executable, there may be no *executable and definition free path* connecting these two in the CFG. This algorithm excludes the influence of such paths on its analysis.

The complexity of this algorithm is O(N+C), where N is the number of nodes in the CFG and C is the number of du-chains in the program.

The drawbacks of this algorithm are :

1. The global value graph may not be as sparse as the du-chain graph on an average, as has been shown in [WeZa85]. Hence, the execution time may suffer in comparison to the simple du-chain graph based algorithm. However, the size of the global value graph grows more slowly than that of the du-chain graph for the same program.

2. The introduction of the identity assignments will increase the number of nodes in this graph, which also has the same effect of increasing the execution time of this algorithm.

2.3.4 Interprocedural constant propagation

The constant propagation algorithms considered so far have ignored the effects of procedure calls. In such algorithms, procedure calls have to be included by assuming that any variable can be modified by this call and so the effect of a call is to wipe out all constant definitions. [CCKT86] describes a method to incorporate procedure calls in the framework of constant propagation by considering their effects less pessimistically.

The method proposed in this paper also uses the constant lattice approach. However, for each call site s and formal parameter f of the called procedure, they maintain a function J_s^f, called *jump function*[4], which is the best approximation, within the lattice, to the value passed to f at s, given the values passed to the formal parameters of the calling procedure p and the values of the global variables of p on entry to it. Along with this they define the *support* of J_s^f to be the exact set of formals and globals of p used in the computation of J_s^f. For each formal f of procedure p, this algorithm uses the jump function J_s^f at each call site s calling p in order to determine the initial lattice value of f upon entry to p. Then the standard constant propagation algorithms are used and, whenever a variable in some support set has its lattice value lowered, then the jump functions are again applied to all formals whose support sets contain this variable. If the new lattice value for any formal is found to be different from what was stored in it, then this formal shall be reexamined in the constant propagation algorithm.

There are three scenarios to determine the jump functions and the support sets at each call site s in a procedure p :

1. If formal f at s is known to be a constant just by examining the source of p, then

[4] So called due to historical reasons.

J_s^f = *constant*, and

support (J_s^f) = $\emptyset$.

2. For any formal f at s that cannot be determined as a function of the formals and globals of p, *e.g.*, when the value passed to f happens to be involved in an I/O read, set

   J_s^f = *bottom*, and

   support (J_s^f) = $\emptyset$.

3. For any formal f at s that is determinable at compile time, but is not known to be a constant as yet, set

   J_s^f = a function of the formals, globals and local constants of p, and

   support (J_s^f) $\neq \emptyset$.

The paper suggests some methods to determine the jump functions in the last case. These are as follows :

1. By considering the effect of all statements which can modify the actual parameters passed at the call sites within a procedure, and assuming all formals and globals of this procedure to have lattice values *bottom* upon entry to it, we can use constant propagation algorithms in each procedure to determine the constancy of the actual parameters passed at call sites in it. This method has the disadvantage that it will detect the effect of only one level of procedure calls and it is forced to consider the formals of procedures to be variable upon entry, since it does not know their possible initial values (an example of this is given in [CCKT86]). This method has to assume that a call to a procedure can potentially change all global variables of the calling procedure as well as all actual parameters that have been passed by reference.

2. Detect cases where variables are simply passed through a procedure to a call site in it. This method can easily be implemented using du-chains and the standard constant propagation algorithms. While this method includes some effects due to multiple levels of calls, it is too conservative in cases where the same parameter is passed as the actual parameter at different call sites in the same procedure. This has been elaborated in [CCKT86].

3. Use information regarding the side effects of procedure calls, such as globals that may be modified during this call, parameters passed by reference being changed due to it, *etc.* This information can be obtained by examining the code contained in each procedure as well as the the parameters passed to the procedures at the call sites. Then by studying the way information is passed through a procedure to call sites within it, we can determine what variables, globals or formal parameters, a procedure may change whenever it is called. The process of collecting this information is called *interprocedural data flow analysis* ([ASU86, pages 653-660]). The details of this analysis scheme has been explained in Chapter 4 of this report. The incorporation of the interprocedural information will make the jump functions dependent on the calling pattern in the program. In this case more specific information can be obtained as to the set of variables that may be modified by a procedure call, and so the jump functions can be made to be conditional on the variables modified by procedure calls. The paper also suggests using *return jump functions* to take into the account the possibility of constant return values by function calls.

Since the value of J_s^f is evaluated each time some element of $support(J_s^f)$ has its lattice value reduced, the time for this algorithm is proportional to the quantity :

$$\sum_s \sum_f time(J_s^f) \mid support(J_s^f) \mid, \text{where}$$

$time(J_s^f)$ is the time to compute the function J_s^f,

s ranges over all the call sites in the program, and

f ranges over all formals that are bound by the call at s.

If the time to compute each J_s^f and the size of each support set are bounded by constants, then the above implies that the total time is proportional to the sum over each call site in the program of the number of parameters passed along it.

The major disadvantage of this scheme is that it needs two levels of constant propagation – one to compute the jump functions and a subsequent one to use these functions to incorporate the effects of procedure calls. Further, the efficiency of this scheme is inherited from that of the basic constant propagation algorithm used. However, this algorithm takes into account the calling patterns in the program in order to do a more intelligent constant propagation than naive algorithms.

2.4 Aliasing

Aliasing is the general name given to the phenomenon wherein two different names may or will refer to the same memory location(s). [ASU86] gives a good in-depth introduction to this subject. Aliasing can arise due to three reasons :

1. Due to reference formal parameters.
2. Due to storage allocation statements such as COMMON and EQUIVALENCE in Fortran.
3. Due to pointers.

We shall discuss all three of them in brief here.

2.4.1 Reference formal parameters

When parameters are passed by reference, the formal parameter of a procedure, after a call is made, points to the same memory location(s) as the actual parameter passed to this formal at the site of this call. However, since there may be several calls to the same procedure, a single formal parameter of a procedure can be aliased to a number of variables. The problem of aliasing, in this case, is to determine the set of variables which each formal parameter of the procedures in the program may be aliased to upon entry to the corresponding procedures. The following Fortran fragment shall serve to illustrate this phenomenon :

```
 1. subroutine A ( f1, f2, f3 )
                    .
 2.         common /block1/ g, h
 3.         integer :: m, n, p
                    .
 4.         f1 = 6
 5.         g = 75
 6.         m = f1
 7.         n = f2
                    .
 8.         call A ( f1, f1, p )
                    .
 9. end
10. program main
                    .
11.         common /block1/ g, h
                    .
12.         call A ( g, 0, -3 )
                    .
13. end program
```

In this example one notes the fact that the call to procedure A at line 12 has the effect of causing formal parameter f1 of A to point to the same memory location as the global variable g. Consequently, on entry to A, f1 and g will be aliases of each other. However, the call at line 8 will cause f2 to point to whatever f1 was pointing to upon entry to A. Although this happens only during the next activation, we have to analyze the static program text. Consequently, we shall make the assumption that both f1 and f2 may point to the same location(s) whenever procedure A is entered. Hence, f2 will also be pointing to g, since at the point of the call at line 8, f1 was pointing to g. This means that the assignment to g at line 5 may cause both f1 and f2 to change. If we were to ignore the aliasing information, then we would, for example, propagate the "constant" value of f1, assigned at line 4, to the use of f1 at line 6, which is erroneous. At the same time it must be noted that f3 has no aliases and so it does not get altered due to changes to any other variable or formal.

The standard method of analyzing such aliases ([ASU86]) is inefficient in that it is a direct method involving the computation of the reflexive and transitive closures of the alias relation. However, the alias relation is *not* transitive, as has been shown in Chapter 3, and in assuming it to be so, this algorithm is excessively conservative.

Cooper ([Coo85]) has identified the aliasing problem as arising in one of two phases :

1. **Alias introduction :** This is the problem of finding where fresh aliases are created, without taking into account the existing ones. There are two case to consider here :

   (a) When the same actual parameter has been passed in two different formal parameter locations at a call site, these two formal parameters become aliased to each other in the called routine.

   (b) When a global variable (of the called routine) is passed as an actual parameter at a call site, this global and the corresponding formal parameter form aliases of each other in the called routine.

2. **Alias propagation :** This is the problem of finding how existing aliases give rise to new ones. This, again, can be identified as arising under two conditions :

   (a) When two aliased entities are passed as actual parameters at the same call site, their corresponding formal parameters also get aliased to each other, within the called procedure.

(b) When a global variable and an item aliased to it (in the calling procedure) are passed as actual parameters at the same call site, then their corresponding formal parameters also get aliased to each other, within the called procedure.

Based on these observations, Cooper ([Coo85]) has given an algorithm to determine these aliases. This algorithm requires time $O(N^2E)$ in the worst case, assuming that the number of formal parameters of any procedure is bounded by some constant, where N is the number of procedures in the program and E is the number of call sites in the program. Of course, this algorithm is no longer as conservative as the one in [ASU86]. However, it's only a straightforward implementation of the alias introduction and propagation rules given above. Cooper, along with Kennedy ([CoKe89]) modified this algorithm in order to implement it in a more efficient manner. This algorithm utilizes several interesting features of the alias introduction and propagation phases in order to make it run in time O(NE). This is the algorithm which we have implemented in this work and the full algorithm has been explained in Chapter 3.

2.4.2 Storage allocation statements

Fortran allows storage allocation statements, *e.g.*, COMMON and EQUIVALENCE statements, which can give rise to aliasing. This is because these statements allow us, directly or indirectly, to force different variables to share whole or part of the data storage allocated to them. Such sharing will also give rise to aliases between the variables that do so. This has been discussed at length in Chapter 3. As an example we shall consider the following Fortran fragment :

```
 1. program main
                .
 2.       integer :: a(10), b, c, d
 3.       equivalence ( a(5), b )
 4.       equivalence ( a(5), c )
                .
 5.       b = 8
 6.       c = 9
 7.       d = b
                .
10. end program
```

Clearly, the variables b and c have to be allocated the same storage location(s) in order to satisfy the two equivalence statements in lines 3 and 4. Consequently after the assignment to c at line 6, b also has to be considered as modified, *i.e.*, b and c are aliases of one another.

Otherwise, we could erroneously propagate the "constant" definition of b at line 5 to its use in line 7.

As a further example, we shall consider yet another Fortran fragment :

```
1. subroutine A ( ... )
                .
2.          common /block1/ a1, b1, c1
3.          integer a1, b1, c1
                .
4.          b1 = c1
                .
5. end

6. subroutine B ( ... )
                .
7.          common /block1/ a2, b2, c2
8.          integer a2, b2, c2, d2
                .
9.          b2 = 99
10.         call A ( ... )
11.         d2 = b2
                .
12. end
```

From the semantics of COMMON statements we can see that variables b1 and b2 should be allocated the same space in memory as each other. In this case the call at line 10 has the effect of changing the contents of the memory location(s) pointed to by b2 indirectly through the assignment to the variable b1 at line 4. Thus these two variables are aliases of one another globally. This knowledge will prevent us from erroneously propagating the "constant" definition of b2 at line 9 to its use at line 11.

[ASU86] gives good algorithms to handle such situations. The algorithms used in this work are adaptations of these.

2.4.3 Pointers

In languages that allow access to memory locations through pointers, there exist a different kind of aliasing. This occurs whenever any number of pointers, within the same scope as a variable, can be pointing to the same memory location(s) as this variable. Under these conditions, the contents of these pointers, which are addresses of the said memory location(s), will be aliases of the this variable.

This work does not handle this type of aliasing, since the final version of Fortran 90 does not have pointers, although the Fortran 8x draft mentions them. A good paper on pointer aliasing is the one by Landi and Ryder ([LaRy92]). In this algorithm arbitrary

pointer references are handled by considering some predetermined integer k and limiting the number of indirections of a pointer that will be considered for the sake of alias analysis to this value. If a pointer has more than this many dereferences then this algorithm shall consider only the first k dereferences. This is a conservative assumption in that it may lead to an overestimate of the set of aliases for the variables and so impede some optimizations[5]. This method is called *k-limiting*.

[5][ASU86] discusses the subject of conservatism in data flow analysis in great detail.

Chapter 3

Alias analysis

In this chapter we discuss the issue of aliasing analysis in Fortran 90 programs and the algorithms used to do this analysis. We shall also look briefly at the actual implementation of these algorithms and the data structures used therein.

3.1 What is aliasing ?

As mentioned in the previous chapter, aliasing is the general name given to the phenomenon wherein two different names either definitely or possibly refer to the same memory location(s) ([ASU86]). As we shall see in the course of this discussion, the two variables that refer to the same memory location(s) need not do so at all times at the same point of the program. The forms of aliasing that can arise in Fortran 90 programs are primarily limited to the following :

1. Aliasing due to reference formal parameter association.
2. Aliasing due to associations of variables in COMMON and EQUIVALENCE statements.

Both these forms of aliasing have been dealt with in this work. We shall discuss these forms of aliasing at length now.

3.2 Aliasing due to COMMON/EQUIVALENCE association

3.2.1 COMMON association

The scenarios involved

In Fortran global variables are declared as belonging to COMMON blocks, with each procedure (function or subroutine) using its own set of names for elements within a COMMON block ([For8x]). This cannot give rise to aliases between data items in *different* COMMON blocks. Neither does it give rise to aliases between data items declared explicitly to be in the *same* COMMON block in *a particular procedure*. But what can happen is that data items in the *same* COMMON block but declared in *different procedures* can become aliased to each other if the storage spaces of these variables overlap within this block. Consider, for example, the following Fortran fragment :

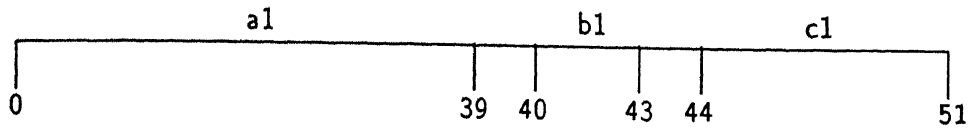
```
subroutine sub1 ( ... )
        .
        .
common /block1/ a1(10), b1, c1(2)
integer a1(10), b1, c1(2)
        .
        .
end

subroutine sub2 ( ... )
        .
        .
common /block1/ a2, b2(11), c2, d2
integer a2, b2(11), c2, d2
        .
        .
end
```

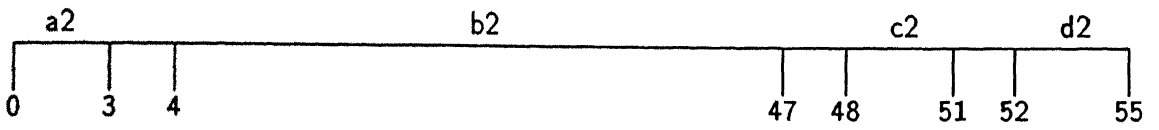
The storage allocation for *block1* according to the two procedures sub1 and sub2 is given in figure 3.1 (assuming integers to occupy 4 bytes each).

Since the storage allocated for the block *block1* is the same set of bytes, we note the following :

- a1 and a2 are aliases of each other, since they share bytes 0 through 3.
- a1 and b2 are aliases of each other, since they share bytes 4 through 39.
- b1 and b2 are aliases of each other, since they share bytes 40 through 43.



(a) according to procedure *sub1*



(b) according to procedure *sub2*

Figure 3.1: Storage allocation for *block1* according to procedures *sub1* and *sub2*.

- c1 and b2 are aliases of each other, since they share bytes 44 through 47.
- c1 and c2 are aliases of each other, since they share bytes 48 through 51.
- d2 has no aliases.

The basic problem here is to determine the sets of global variables for each COMMON block in the program, such that variables in each set will be pointing to the same memory location(s).

3.2.2 EQUIVALENCE association

The scenarios involved

The EQUIVALENCE statement in Fortran 90 allows us to specify to the compiler that we want different variables to share whole or parts of the memory storage allocated to them. Therefore, it is the most direct method of introducing aliases in Fortran ([For8x]).

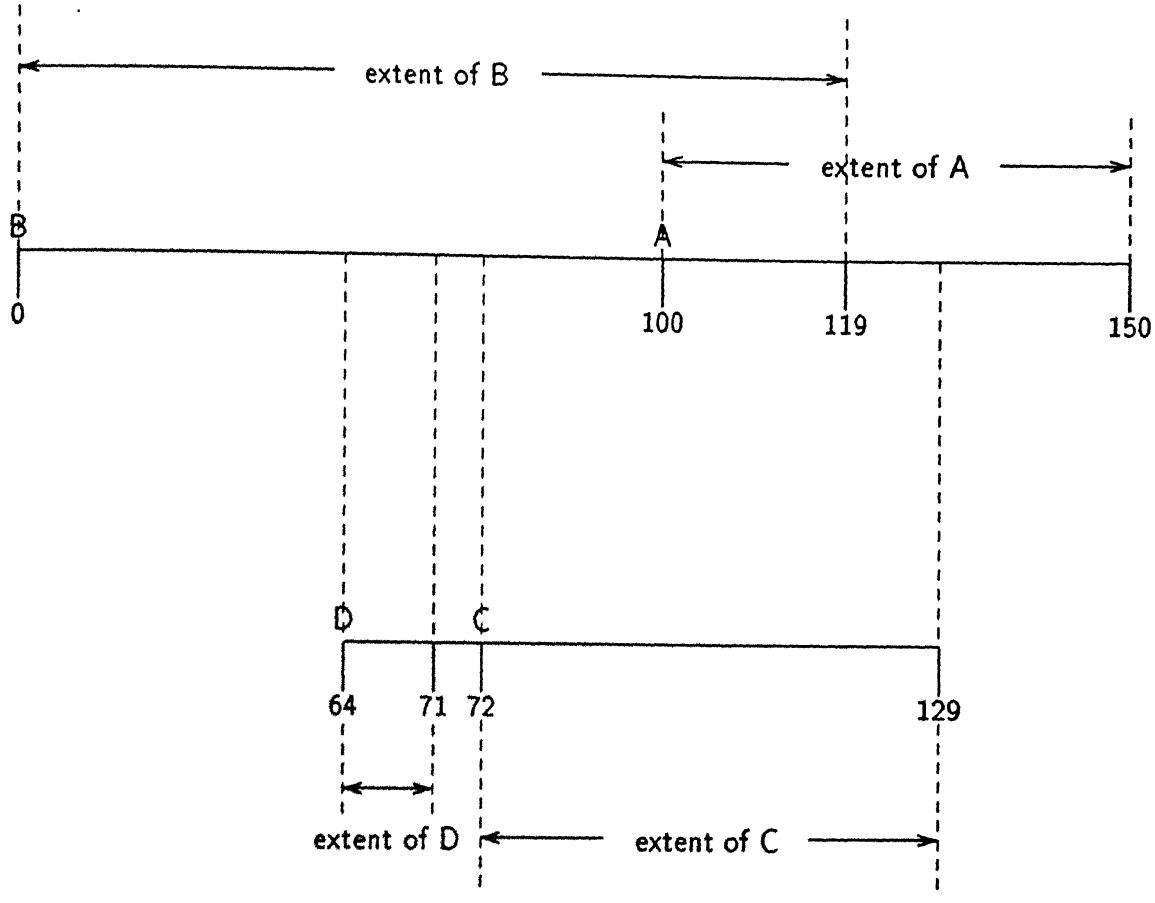


Figure 3.2: Overlapping of storage areas due to EQUIVALENCE.

A bit of reflection tells us that, through the EQUIVALENCE statement, Fortran allows us to specify that some variable A should be allocated space starting some, say, *offset-FromB* number of bytes from the starting location of some other variable B (although this information is not specified as such, the compiler can derive this information anyhow).

This can give rise to aliases whenever such placement of storage allocation for A causes it to overlap with the storage for other variables, which have also been specified to be at some fixed offset from B.

Consider the scenario shown in figure 3.2. Here the storage area allocated to A overlaps with that for C. Hence A is an alias of C also, besides being an alias of B. There are two important points to be noted, in connection with this scenario :

- C need not have been specified to be in EQUIVALENCE to B at an offset of 72 bytes from it. Rather we could have had some variable D specified to be in EQUIV-

ALENCE to B at an offset of 64 bytes from it and, in turn, C being specified to be in EQUIVALENCE to D at an offset of 8 bytes from it.

- In the example of figure 3.2, A has become an alias of B. However, this is true only because A and B overlap to some extent and *not* because the compiler was requested to put A 100 bytes beyond the start of B. If these two variables did not have any overlap in their storage allocations, we would have no alias between them[1].

However, apart from the above mentioned scenarios we can have another kind of aliasing. This is the case where an element is in EQUIVALENCE to some item in some COMMON block, either directly through an EQUIVALENCE statement containing both these two variables, or indirectly through EQUIVALENCE to some variable which, in turn, is in EQUIVALENCE to an item in COMMON. Under these conditions, this variable and, indeed, all variables directly or indirectly in EQUIVALENCE to it, shall go into this COMMON block, and will become global variables.

The problem here is to find the set of variables, globals or locals, that are aliased to each other due to any one of the scenarios mentioned here.

3.2.3 Two important properties

At this point it's worthwhile to consider two important properties of aliasing due to COMMON/EQUIVALENCE association :

1. Aliasing due to COMMON/EQUIVALENCE association is *not* transitive, although it is both reflexive and commutative[2]. This means that aliasing is not an equivalence relation, although the name EQUIVALENCE seems to indicate it to be so. A simple example is the allocation shown in figure 3.3.

   Clearly B is aliased to A and A to C. But B is *not* aliased to C, since B and C do not have any storage area in common.

[1] Actually, such a scenario cannot occur in Fortran, because the only way one can specify A to be put 100 bytes beyond B is for B to be an array and an EQUIVALENCE statement of the type EQUIVALENCE (B(26), A) to have been used in the program (assuming B to be an array of integers, which occupy 4 bytes each). In this case, obviously, B(26) must be an element of B and hence A and B must overlap.

[2] Actually, this is true for aliasing due to reference formal parameter association as well, as we shall see in Section 3.3.

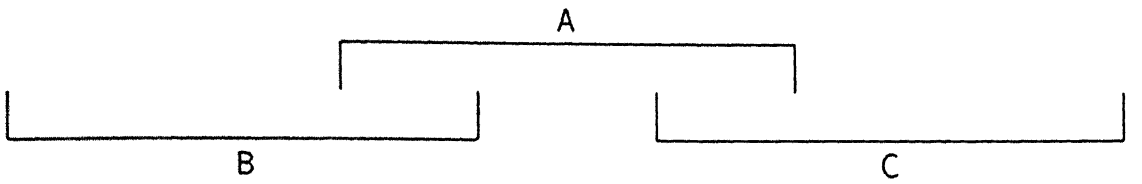


Figure 3.3: Demonstrating non-transitivity of COMMON/EQUIVALENCE aliasing.

2. Aliasing due to COMMON/EQUIVALENCE association is *definitive* aliasing, i.e., the alias relation between two data items holds at all times and places in the program, wherever they are in scope. This is so because this type of aliasing arises due to definite overlap in the storage areas for the two items which are aliased and this overlap always holds true. As we shall see in Section 3.3, aliasing due to reference formal parameter binding is *not* definitive and is termed as *maybe* aliasing.

3.3 Aliasing due to reference formal parameter association

In Fortran 90 functions and subroutines pass parameters by reference. This means that changes made to the formal parameters shall be reflected in the actual parameters themselves, unless these are constants or expressions that do not form what is called an *l-value*. An *l-value* is the value that represents a location rather than the value stored therein. An expression, for instance, does not form an *l-value*, while a single variable, an array element, *etc.*, can do so.

The formal parameter of a procedure and the corresponding actual parameter passed by reference at a call site, will point to the same memory location(s) upon entry to this procedure. However, since there may be several calls to the same procedure, a single formal parameter of a procedure can be aliased to a number of variables. The problem of aliasing, in this case, is to determine the set of variables to which each formal parameter of the procedures in the program may be aliased to upon entry to these procedures.

3.3.1 The scenarios involved

We shall now elaborate the methods of formation of aliases due to formal parameter association, as had been alluded to in the previous chapter. Upon entry to a procedure each

of its formal parameters may be aliased to one or more global variables. Also it may be aliased to other formal parameters of this procedure. Aliasing of this kind can arise due to either of two reasons :

1. Alias introduction, and
2. Alias propagation.

Both these two means of aliasing take place at call sites only. We shall discuss both these methods now.

1. **Alias introduction** : This is the direct method of forming fresh aliases. At a call site s a formal parameter f_1 of the called procedure p can be :
   (a) aliased to a global variable x if x is passed as the actual parameter for f_1 at s, or
   (b) be aliased to another formal parameter f_2 of p if either
      i. the same variable x is passed as the actual parameter in locations f_1 and f_2, or
      ii. two variables which are aliased under COMMON/EQUIVALENCE association are passed as actual parameters in locations f_1 and f_2, or
      iii. if some formal parameter f_0 of the calling procedure q is passed as the actual parameter in the location f_1, while some global variable x that is aliased to f_0 (in q) through formal parameter association or which is in COMMON/EQUIVALENCE aliasing with such a global variable is passed as the actual parameter in location f_2.
2. **Alias propagation** : At a call site s, aliasing of the formal parameters of the calling procedure q can lead to new aliases being generated in the called procedure p also. This can arise due to one of two main reasons :
   (a) A global variable x, which is an alias of a formal parameter f_1 of q, will be passed on as an alias of a formal parameter f_3 of p, if f_1 is passed as the actual parameter in location f_3 at the call site s.
   (b) If f_1 and f_2 are formal parameters of q and are aliased at this point, and these are passed as actual parameters in locations f_3 and f_4 of p, respectively, at the call site s, then f_3 and f_4 become aliases of each other upon entry to p.

3.3.2 Three important properties

We shall examine some important properties of this type of aliasing :

1. Akin to COMMON/EQUIVALENCE aliasing, aliasing due to reference formal parameters is also commutative and reflexive, but *non*-transitive, though this latter property arises due to the path sensitivity of this aliasing operation. Two variables (whether formals or global variables) shall be deemed to be aliases of each other only if they get bound *along the same sequence of call statements.* Consequently, if we have a case where A is aliased to B and B is aliased to C, then A is aliased to C if and only if we can independently arrive at this conclusion and *not* just by applying transitivity. A concrete example of this is given in the following Fortran fragment :

```
 1. subroutine A ( f1, f2 )
                  .
 2.         common /block1/ g, h
 3.         integer :: m, n
                  .
 4.         call B ( m, n )
 5.         call A ( n, h )
 6.         call A ( f2, f2 )
                  .
 7. end
 8. subroutine B ( f3, f4 )
                  .
 9.         common /block1/ g, h
10.         integer :: x
                  .
11.         call A ( x, x )
                  .
12. end
13. program main
                  .
14.         common /block1/ g, h
                  .
15.         call A ( g, 0 )
                  .
16. end program
```

Here g and h are globals (being in COMMON). The call at line 15 causes g to get aliased to f_1 upon entry to procedure A. The call at line 11 causes f_1 and f_2 to get aliased to each other on entry to procedure A. But we cannot say that g is an alias of f_2, since we cannot draw this conclusion independently. On the contrary, the call at line 5 makes h an alias of f_2 on entry to procedure A and the one at line 6 makes f_1 an aliases of f_2 on entry to A. This shall make h an alias of f1 as well.

These two results are neither surprising nor contradictory. In no activation of the recursive routine A will both f_1 and f_2 simultaneously represent the same memory location(s), as represented by g. However, after the call to A at line 5, the new activation of procedure A has f_2 pointing to h. When this activation calls itself at line 6, this latest activation will have f_1 pointing to what f_2 was pointing to, in the previous activation, *viz.*, h. Moreover, f_2 will also be pointing to this same location. Hence the conclusions.

2. Unlike COMMON/EQUIVALENCE aliasing, formal parameter aliasing is only *maybe* aliasing, i.e., the aliasing relation may not hold at all times at a given place in the program. This is brought out by the preceding example itself where f_1 and f_2 are definitely not aliased on entry to A when the call at line 15 is make. However, after the call at line 6, f_1 and f_2 are definitely aliased to each other. Unfortunately, we have to analyze the static text and not the program in execution. So conservatism demands that we consider all possible alias pairs at the same point of the program, irrespective of whether they will all be active at the same time at that point or not, and assume that such aliasing relation may or may not hold true at that point. This shall lead to the concept of ambiguous definitions, as we shall see in the next chapter.

3. Despite all that talk about non-transitivity of the alias relation, we do have at least one notable *partial* exception to this. Consider the case where A is aliased to B and B is aliased to C, where one of these two aliasings is due to COMMON/EQUIVALENCE association while the other is due to formal parameter association. In this case we will have a *single level* of transitivity between A and C and the resulting aliasing will be of the *maybe* type. This means that we can apply this transitivity *only* if A is aliased to B and B to C without considering any transitivity in the aliasing between A and B or B and C. To realize why this is so, we consider, without loss of generality, that A and B are aliased to each other due to formal parameter association, while B and C are aliased to each other due to COMMON/EQUIVALENCE association. Then each time A represents the same memory location(s) as B, it may also represent the same memory location(s) as C, since B and C overlap in their storage areas. However, it's important to bear in mind that one must not carry this transitivity to more than one level, since that would imply transitivity of both types of aliasing *separately*, which is

not true.

3.4 The alias analysis algorithms and their implementation

We now discuss the algorithms that have been used in this work in order to analyze the aliasing patterns in the program, under all three types of aliasing mentioned so far. We shall also briefly look at the issues related to the implementation of these algorithms.

3.4.1 COMMON association aliasing

The algorithm

In order to determine this type of aliasing in the presence of arbitrary COMMON declarations for any number of COMMON blocks and procedures, we need a systematic method. The algorithm used to achieve this consists of 2 phases :

Phase I :

This phase gathers information about the contents of each COMMON block on a per procedure basis. The algorithm is an adaptation of the one given in [ASU86, pages 446-448, with the modifications suggested in pages 452-454]. The algorithm given therein was augmented in order to extract the aliasing information from the memory allocation performed by this algorithm for each COMMON block. The augmented algorithm is, essentially, as given in figure 3.4.

The EQUIVALENCE analysis operation mentioned in step 9, is the analysis of EQUIVALENCE statements, which will be explained in Section 3.4.2. The augmentation to this algorithm, *viz.*, step 8, will take a *total* time proportional to the sum of the squares of the number of items in each COMMON block, since items are inserted in the alias list in some order. However, if we make the assumption that the number of COMMON blocks is small and the number of items in each block is also small, then this term should not have too much of an influence in the total time complexity of this algorithm.

Phase II :

After phase I, we have the alias list for each COMMON block containing all variables in all procedures that are in this block either directly (due to COMMON statements containing these variables) or indirectly (due to their being in EQUIVALENCE to some variable in

1. For each COMMON block in the program, set the alias list pointer to NULL.
2. For each procedure p do {
3. Set the variable chain for each COMMON block used in this procedure to NULL.
4. For each statement of the form common /*blockname*/ $a_1, a_2, ..., a_n$ do {
5. Append $a_1, a_2, ..., a_n$, in that order, to the variable chain for the COMMON block *blockname*.
   }
6. For each COMMON block *blockname* used in procedure p do {
7. Scan the variable chain for this block assigning offsets from 0 onwards and, after each variable, increment the offset by the number of bytes required for that variable.
8. Also, put each variable in the alias list for this block, in which elements are inserted in ascending order of offsets in the COMMON block.
   }
9. Do EQUIVALENCE analysis for for this procedure p.
10. For each COMMON block *blockname* used in procedure p do {
11. Adjust the size of this COMMON block for accommodating elements due to EQUIVALENCE.
    }
}

Figure 3.4: COMMON association alias algorithm – Phase I.

this COMMON block – see Section 3.4.2 for more details on this). This phase shall finally determine the sets of alias pairs that arise due to COMMON aliasing. The algorithm is as given in figure 3.5.

The above two phases give us the full set of aliases arising due to data items in COMMON blocks, either directly or indirectly (as mentioned earlier). The total time complexity of this phase is, again, the same as what was mentioned about step 8 of Phase I, since the same set of lists are being processed and each list is scanned, albeit partially, for each item on it. The correctness of this phase in determining the required set of aliases is guaranteed by the ordering of these alias lists and the test at step 4, which imply that only elements

```
1. For each COMMON block blockname do {
2.      For each item x in the alias list for this block having offset offsetOfX in this
        block do {
3.          For each item y in the rest of the list ( beyond x ) having offset offsetOfY
            in this block do {
4.              If offsetOfX + number of bytes in x > offsetOfY then
5.                  Make x and y aliases of each other.
            }
        }
    }
```

Figure 3.5: COMMON association alias algorithm – Phase II.

which do have some overlap in their memory allocations are taken to be aliases of each other.

Implementation notes

We maintain a global linked list of information structures, having one entry for each COMMON block in the program. This allows us to access these structures in all the procedures in which a COMMON block may be used.

The alias list mentioned in step 1 of phase I is simply a linked list of entries arranged in ascending order of their offsets in the COMMON block. The variable chain is also a linked list of the symbol table entries of the variables that belong to the COMMON block.

Steps 1 and 3 of phase I cannot be done as has been indicated. This is so because the chaining of variables is done during the parsing stage itself, for the sake of efficiency. So these steps are done as and when the first COMMON statement involving a particular block is encountered, in the entire program in the case of step 1, and in each procedure in the case of step 2.

Before step 10 of phase I, we assume that EQUIVALENCE analysis, for the procedure being considered, to have been completed, as mentioned in step 9. This has been explained in the Section 3.4.2. Further, the block size adjustment mentioned in step 11 has also been

elaborated there.

We define variables to be global variables if they belong to some COMMON block, either directly through some COMMON statement or indirectly through EQUIVALENCE to some element in COMMON. For each global variable we keep a bit vector *globalVariableAliasVector*, in a structure of type `aliasInfoType`, which has 1 bit for each global variable in the program. For each global variable we set the bits corresponding to all global variables to which it is aliased by this algorithm. Of course, we ordinally number all globals in the program prior to assigning bits to each of them. An array *GlobalVariableMapArray* gives us the mapping from a number to the symbol table entry for the corresponding variable.

3.4.2 EQUIVALENCE association aliasing

The algorithm

The algorithm used in analyzing the EQUIVALENCE statements for aliasing patterns is an adaptation of the algorithm given in [ASU86, pages 448-454, including the two optimizations mentioned in page 451]. This algorithm had to be augmented because it was not an alias analysis algorithm, but merely a storage allocation determination algorithm.

The algorithm tantamounts to creating EQUIVALENCE trees with each element belonging to a tree and storing the offset from its parent provided such a dependency has been either specified in the program or can be derived from the ones specified. Initially we assume each variable to be in a tree by itself. Further, if a variable is in COMMON, then we shall record this information by storing the name of this COMMON block and the offset of this variable in it. This assumes the determination of offsets within the COMMON block to have been done already, which is true, since it would have been done in step 7 of phase I of the COMMON analysis algorithm given in Section 3.4.1. Furthermore, it also stores two values - *low* and *high*, initially set to 0 and one less than the number of bytes required for this variable, respectively. This last pair of information is needed in order to determine the size of the EQUIVALENCE block, *viz.*, the sequence of memory locations needed to accommodate all items in an equivalence tree. This will be useful in determining the size of the COMMON block of which this EQUIVALENCE block may be a part.

The algorithm for analyzing EQUIVALENCE statements is given on a per procedure basis for each procedure p in figure 3.6. This is the EQUIVALENCE analysis alluded to in

step 9 of phase I in the algorithm for COMMON analysis in Section 3.4.1.

The algorithm to compute the aliases due to EQUIVALENCE without any variables in COMMON is given in figure 3.7.

This algorithm gives us the full set of aliases arising due to EQUIVALENCE association, *excluding* those for elements in COMMON blocks. It is not too difficult to see from the similarity of this algorithm with the one used in Phase II of COMMON association alias analysis given in Section 3.4.1, that their time complexities and correctness criteria are the same.

Implementation notes

The symbol table entry for each variable has an entry *equivInfoPtr* which points to a structure of type `equivInfoType`. This pointer is NULL for items not in EQUIVALENCE. If this item is the root of an EQUIVALENCE tree, then we have the *equivRootInfoPtr* field in this structure being non-NULL and pointing to a structure of type `equivRootInfoType`. This structure contains information like the name of the containing COMMON block, *etc.*, which are specific to the root of an EQUIVALENCE tree.

One important point to note is that EQUIVALENCE statements in Fortran are of the type :

$$\text{EQUIVALENCE } (var_1, var_2, var_3, ..., var_n).$$

In order to satisfy the requirements of step 1 of this algorithm, we consider this statement to be a number of statements of the form :

$$\text{EQUIVALENCE } (var_1, var_2)$$
$$\text{EQUIVALENCE } (var_1, var_3)$$
$$\vdots$$
$$\text{EQUIVALENCE } (var_1, var_n)$$

1. For each equivalence statement requiring A to be set at *offsetFromB* number of bytes from B do {
2. Get the following values :
   P = the root of the equivalence tree containing A.
   distFromP = offset of A from P.
   Q = the root of the equivalence tree containing B.
   distFromQ = distance of B from Q.
   /* We perform the path compression optimization given in [ASU86, page 451] while getting to P and Q. */
3. If P = Q then {
4. If *distFromP* - *distFromQ* $\neq$ *offsetFromB* then error;
   }
5. else {
6. Use the *number of descendents* method given in [ASU86, page 451], to find the root node to be make a parent of the other one. Let it be Q.
7. Make Q a parent of P.
8. Set offset of P from Q to
   newDist = *distFromQ* - *distFromP* + *offsetFromB*.
9. Set the following :
   low(Q) = *min* (low(Q), low(P) + *newDist*)
   high(Q) = *max* (high(Q), high(P) + *newDist*)
10. If P has a COMMON block name and offset in it stored, then store this name in Q and store in Q its offset in this COMMON block to be the value :
    offset of P in this COMMON block - *newDist.*
    }
}

Figure 3.6: EQUIVALENCE analysis algorithm.

```
1. For each procedure p do {
2.      For each variable r in p which is the root of an EQUIVALENCE tree do {
3.          If r does not have a COMMON block name stored in it
            then
4.              set the alias list pointer in r to NULL.
        }
5.      For each variable x in p which is some EQUIVALENCE tree rooted at some
        variable r
        do {
6.          Check r to see if this tree has an item in COMMON.
7.          If yes then {
8.              Put x in the alias list for this COMMON block, with its offset in
                this block being the offset of r in it + the offset of x from r.
            }
            else {
9.              Put x in the alias list for r, with its offset being its offset from r.
            }
        }
10.     For each variable r which is the root of an EQUIVALENCE tree which
        has no item in COMMON do {
11.         For each item x in the alias list for the EQUIVALENCE block containing
            r, having offset offsetOfX in this block do {
12.             For each item y in the rest of the list ( beyond x ) having offset
                offsetOfY in this block do {
13.                 If offsetOfX + number of bytes in x > offsetOfY then
14.                     Make x and y aliases of each other.
                }
            }
        }
    }
```

Figure 3.7: EQUIVALENCE association alias algorithm.

Then, for each statement of the form EQUIVALENCE (var_1, var_i), we find out the offset of var_1 (remember, this may be an array element or a structure member) from the starting location of its parent variable (*i.e.*, the variable itself, if it's a scalar variable, or the parent array or structure, if it's an array element or a structure member). We do the same for var_i. Then we can set,

A = parent variable of var_1,

B = parent variable of var_i, and

offsetFromB = offset of B from its parent variable - offset of A from its parent variable

in step 1 of the algorithm for analyzing EQUIVALENCE statements.

We define local variables to be those which are neither in any COMMON block nor in EQUIVALENCE with any variable in any COMMON block. For each local variable we have a bit vector *equivLocalAliasVector*, again stored in the `aliasInfoType` structure for this variable. This vector has as many bits as the number of local variables in the program. In order to alias another local variables to this one, we simply set its corresponding bit in this vector. Of course, akin to the globals, we ordinally number the entire set of local variables in the program, prior to assigning these bits. An array *LocalVariableMapArray* maps these numbers to the corresponding symbol table entries of the local variables.

3.4.3 Reference formal parameter association aliasing

The algorithm

The two scenarios given in Section 3.3.1 not only give us the complete set of aliases arising out of reference formal parameter association, but also suggest an algorithm to glean this information from the program. Cooper ([Coo85]) has propounded exactly such an algorithm (minus the inclusion of the effects due to COMMON/EQUIVALENCE aliasing). However, Cooper, along with Kennedy, subsequently modified this algorithm in order to make it run faster ([CoKe89]). We have implemented the latter algorithm for the 2-level scoping case (since, Fortran 90 has doesn't have multi-level scoping), with some modification to take into account the effects of COMMON/EQUIVALENCE aliasing. It must be noted that because of this reason, COMMON/EQUIVALENCE alias analysis should have been done prior to formal parameter analysis. The modified algorithm proceeds in 3 phases.

We shall now discuss these phases. However, before looking at the algorithms, we look at some definitions which will be needed in understanding these algorithms :

1. The *formal parameter binding graph* β is defined to be a graph with one node for each formal parameter in the program. There is an edge from the node corresponding to formal f_1 of procedure p to the node corresponding to formal f_2 of procedure q (may be the same as p) iff f_1 is passed as the actual parameter in location f_2 at some call site in p calling q.

2. The *formal pair binding graph* π is defined to be a graph having one node for each pair of formal parameters (f_1, f_2), where f_1 and f_2 belong to the same procedure p. There is an edge from the pair (f_1, f_2) of procedure p to the pair (f_2, f_4) of procedure q, if there is some call site s in p, calling q, where f_1 and f_2 are passed as actual parameters at locations f_3 and f_4, respectively.

3. A graph is said to be *topologically sorted* when each node in this graph is put in such an order that it always comes only *after* all its predecessors in it, though not necessarily immediately after them.

The details of these three phases are given in figures 3.8, 3.9 and 3.10.

The above three phases give us the full set of formal parameters aliased to global variables in each procedure and the global variables and other formal parameters of the same procedure aliased to any of its formals. An important point to note in phases II and III is that the pairs of formal parameters being considered are *unordered* pairs.

The augmentation that we have done to this algorithm amounts to the use of COMMON/EQUIVALENCE aliases of variables wherever any variable is involved in the alias analysis. The definitive nature of this type of aliasing implies that this is justified.

Implementation notes

Phase I. Global alias computation.

The binding graph β was explicitly constructed by having a structure pointed to by the pointer *sccInfoPtr* in the `formalInfoType` structure maintained for each formal parameter. A pointer to this latter structure is kept in the symbol table entry for this formal parameter.

1. Construct the binding graph β.
2. Find the strongly connected components (SCCs) of β.
3. For each SCC c of β do {
4. Replace c with some node in it as the representative node, to get a reduced graph with edges from one nodes in this SCC to nodes in other SCCs being replaced by edges from this representative node to the representative nodes of those SCCs.
   }
5. For each node x in this reduced graph do
6. $A(x) = \emptyset$.
7. For each call site s do {
8. For each global variable g passed to formal f at s do {
9. Let r = representative of the SCC containing f
   $A(r) = A(r) \cup \{g\}$
   }
   }
10. Topologically sort the reduced graph.
11. For each node f in this reduced graph do in forward topological order {
12. $A(f) = A(f) \cup (\bigcup_{p \in predecessor(f) \text{ in it}} A(p))$
    }
13. For each SCC c of β do {
14. For each vertex f in c do {
15. Let n = representative of c
    $A(f) = A(n)$
    }
    }
16. For each procedure p do {
17. For each formal parameter f of p do {
    For each global variable $g \in A(f)$ do
    Make f an alias of g in procedure p.
    }
    }

Figure 3.8: Formal parameter alias analysis – Phase I.

1. Construct the pair binding graph π.
2. Initialize a worklist W to $\emptyset$.
3. For each call site s in the program do {
4. If some variable x is passed as the actual parameters in two different formal parameter locations f_1 and f_2, of the called procedure p, then
5. Mark pair (f_1, f_2) and put it on W.
6. If some variable x is passed as the actual parameter in location f_1 of the called procedure p, and some variable y, which is an alias of x by COMMON/EQUIVALENCE association, is passed in location f_2 of p, then
7. Mark pair (f_1, f_2) and put it on W.
8. If a formal parameter f_0, of the calling procedure q, is passed as the actual parameter in location f_1 of p, then {
9. For each global variable g $\in$ A(f_0) do {
10. If g is passed as the actual parameter in location f_2 of procedure p at s, then
11. Mark pair (f_1, f_2) and put it on W.
12. If some global variable h, which is an alias of g by COMMON/EQUIVALENCE association, is passed as the actual parameter in location f_2 of procedure p at s, then
13. Mark pair (f_1, f_2) and put it on W.

}
}
}
14. While the worklist W $\neq \phi$ do {
15. Remove a pair p from W.
16. For each node q which is a successor of p in π do {
17. If q is unmarked then
18. Mark q and put it on W.

}
}

Figure 3.9: Formal parameter alias analysis – Phase II.

```
1. For each procedure p do {
2.         For each formal parameter f1 of p do {
3.                 For each global variable g ∈ A(f1) do {
4.                         Make g an alias of f1 in procedure p.
                   }
5.                 For each formal parameter f2 of p that forms a marked pair (f1, f2) in π
                   do {
6.                         Make f2 an alias of f1 in procedure p.
           }
   }
```

Figure 3.10: Formal parameter alias analysis – Phase III.

The algorithm for finding the SCCs of β was a straightforward implementation of the algorithm given in [AHU80, pages 189-195]. This algorithm selects a special node in each SCC, called the root of that SCC, and stores a pointer to this root in all nodes belonging to this SCC. These root nodes naturally select themselves as the representatives of their containing SCCs. As an optimization for space, the A vectors are explicitly stored only in the roots of the SCC, since all nodes in an SCC will have the same value for this vector, anyway. Furthermore, we store the ordinal number of the formal parameter corresponding to the root of an SCC in each node of the SCC. A function is provided that takes any formal parameter number as argument and returns its A vector by peeking into the entry of it's corresponding root. This allows us to omit steps 13 through 15. The A vectors are simply bit vectors with one bit for each global variable in the program.

The topological sort of step 10 was again an almost word for word implementation of the program given in [HS80, pages 303–307].

At this stage it is worth it to consider the fact that while [CoKe89] mentions that both β and the reduced graph can be multigraphs, this fact is of no use to us in terms of implementation. Hence we have prevented both these graphs from being multigraphs by avoiding multiple edges between the same pair of nodes as well as self-loops.

The *union* operation in step 12 is merely a bitwise OR operation of the bit vectors. In fact, throughout this implementation, all set operations have been performed using bitwise

operations on bit vector representation of these sets.

An interesting aspect of global variables in Fortran is that these have to be explicitly declared in each procedure in which they are used. Consequently, each global variable has an entry in the symbol table of the procedure containing it. Therefore the set of aliases to a global variable is stored as a bit vector, *formalAliasVector*, in a structure of type `aliasInfoType`, a pointer to which is stored in the symbol table entry for this global variable in this symbol table. This bit vector has as many bits in it as the total number of formals in the program. Actually, this should have been limited to the number of formals in the procedure containing this global variable. This can be optimized for later, although it needs a more contrived mapping from ordinal numbers to formal parameters.

Phase II. Formal pair analysis.

The implementation of step 1 was a little tricky, considering the storage requirements that would arise from a complete building of the pair binding graph π. In view of the fact that we expect only a few aliased pairs of formals per procedure[3], the final implementation was to consider the pair (f_1, f_2) as an entry in a linked list of all formals that form pairs with whichever of f_1 and f_2 has a lower ordinal number among the two. This list is stored in the `formalInfoType` structure for this lower numbered formal parameter. When we refer to a pair (f_1, f_2), it represents the corresponding entry in this linked list for the formal which has a lower ordinal number. Furthermore, the total number of such entries in all such linked lists put together, equals the sum of the number of formal pairs marked and the number of unmarked formal pairs that have edges (in π) leading to them. Step 12 is not carried out explicitly. Instead, it is done along with the marking phase. When a pair is required to be marked, we search for it in the appropriate linked list. If it isn't found in it, then we create it. Then it is marked. If an edge has to go from one pair to another, either or both are created if they are not already present in the relevant linked lists.

The worklist was implemented merely as a queue of pointers to the entries of the said linked lists. The implementation of the rest of this algorithm is quite straightforward.

Phase III. Determining formal parameter alias sets.

In the `aliasInfoType` structure for each formal in the program, we use a bit vector *globalVariableAliasVector*, having as many bits as the number of global variables in the

[3]In any case, the number of formal parameters in each procedure is expected to be small, on an average.

program, to store the set of global variables aliased to it. Over and above this we use the same vector as was used to store the formal parameter alias information at the end of phase I, *viz.*, *formalAliasVector*, to store the set of formals aliased to particular formal parameter.

It would be very costly to implement step 5 as being nested inside the loop at step 2. This is so because for any particular formal parameter f, we may not have all formals, forming marked pairs with it, in the linked list mentioned previously (due to the ordering mentioned there). Instead, the complete list of formals is traversed and each linked list is traversed in turn. When we come to a pair (f_1, f_2), we set the bit corresponding to f_1 in the *formalAliasVector* bit vector for f_2 and *vice versa* (due to the unordered nature of the pairs).

During formal parameter alias analysis, it is often necessary to traverse each call site in a procedure. To facilitate this we have made linked lists of the syntax tree nodes of all call sites in each procedure and stored this linked list in the global structure for this procedure. This representation comes in useful later, during inter-procedural analysis, when we can use this as the adjacency list for each procedure in the call graph.

Chapter 4

Reaching definitions and constant propagation

One of the most important pieces of information that is required to be collected for any type of optimization is the reaching definitions information. In doing this analysis, we have implemented the classical algorithms, dealt at length in [ASU86], albeit with some modifications necessary to incorporate aliasing information as well as some modifications for overcoming some implementation related problems. In this chapter we shall be looking at the algorithms for reaching definitions analysis and constant propagation as well as some implementation related issues about these algorithms.

4.1 Control flow graph construction

The program is input to the optimization phase in the form of a syntax tree and the output of this phase is also expected to be a syntax tree, since this has to be input to the vectorizer. Hence, the CFG has been so constructed as to allow the syntax tree structure to be maintained at each stage of the optimization process. Each node in the CFG represents a source level statement and contains a pointer to root node of the syntax subtree for this statements. Only statements, like assignment statements, call statements, do loop constructs, if then else constructs, *etc.*, have been included in this CFG. This has been done in order to prevent too many nodes in it. We feel that this clearly captures the control flow in the program with an eye to retaining the syntax tree structure after each transformation.

CFG nodes corresponding to goto statments have an edge nodes corresponding to their targets. In the do loop structure, extra CFG nodes have been introduced in order to maintain the use/definition semantics of the loop variables, as specified in [For8x, pages 8-7 to 8-9].

As mentioned in Section 2.1.1, all data flow information, in particular the reaching definitions information, has been computed at the basic block level only. The basic blocks have been implicitly constructed by marking the nodes in the CFG as headers (first nodes) of basic blocks or trailers (last nodes) of the blocks wherever applicable. To traverse a full basic block, we start from its header node and follow its successors (of which we will have only one at each stage, as long as we are within the same basic block) until we reach its trailer, which may be the same as the header node.

At this point it must be mentioned that we construct one CFG for each procedure in the program and keep a pointer to its root node in the global structure holding information about this procedure.

4.2 Uses and definitions

4.2.1 What constitutes uses and definitions

A variable is considered to be definitely used in a statement if there is a textual reference to that variable or to any of its definite aliases in it. Similarly, a variable is considered to be definitely defined by a statement if this variable or any of its definite aliases is assigned a new value in this statement. However, one must consider the case of array references, either as array elements or as array sections or as the entire array itself, differently from scalar variables. This is because a reference to an array, either as a use or as a definition, can mean a reference to parts of the array as well. In this work we shall make the conservative assumption that whenever an array is referenced, either wholly or a part thereof, it will be considered to be a reference to the *entire* array. A more rigorous data flow analysis scheme should be used to determine the portion of the array being referenced and consider this statement as a reference to this portion only. However the corresponding data flow analysis scheme will be more involved while the efforts generally won't be paid back appreciably.

4.2.2 Ambiguous and unambiguous uses/definitions

Uses

A use of a variable x is said to be *unambiguous* if its value is definitely needed during execution of that statement. If, however, we cannot say definitely whether its value will be needed or not, then this statement will be termed to be an *ambiguous* use of this variable. We need to distinguish between these two types of uses because only unambiguous uses can be folded, during constant propagation[1]. Since there may be too many ambiguous uses, we store these two informations separately, in order to reduce the time needed to access the unambiguous uses. However, information on both these two types of uses is necessary because in the subsequent optimization phases, such as live variable analysis, dead code elimination, *etc.*, one must know about all uses of a variable – be it ambiguous or unambiguous. These two types of uses can be more easily understood through the following Fortran fragment :

```
 1. subroutine A ( f1, f2 )
 2.       integer :: m, n
                .
 3.       m = f1
                .
 4. end

 5. subroutine B ( f3, f4 )
 6.       common /block1/ g, h
 7.       integer :: y
                .
 8.       call A ( g, y )
                .
 9. end

10. program main
11.       common /block1/ g, h
12.       integer :: x
                .
13.       call A ( x, 89 )
                .
14.       call B ( -1, x+4 )
                .
15.       write ( x )
                .
16. end
```

We can see here that statements 3 and 15 definitely use the values of the variables f1 and x, respectively. Hence these two are unambiguous uses of their corresponding variables. However, the call at line 8 passes the global variable g as the actual parameter for formal

[1] And, that too, only under certain conditions, as we shall see later.

f1 of procedure A. This procedure, in turn, uses its formal f1 in line 3. This use of f1 by A translates to the use of g by B. Consequently, the call to procedure B at line 14 causes the corresponding statement to be an ambiguous use of g. Note that this is considered to be an ambiguous use of g, since we cannot say whether the call to procedure A at line 8 will always be executed or not. Control flow within A can cause this statement to be bypassed sometimes. Since, g will get used, via f1, only if this call comes through, we're forced to assume the call at line 14 to be an ambiguous use of g. In a similar fashion, the call at line 13 is an ambiguous use of x. This is ambiguous because the use of f1 in line 3, which will translate to a use of x at line 13, may not take place since the statement at line 3 may be bypassed due to control flow within A. For the same reason the use of g in the call statement at line 8 is also an ambiguous one.

Definitions

A definition of a variable x is said to *unambiguous* if after executing that statement, x will definitely have a new value assigned to it (not necessarily different from the old one). If such an assignment may or may not have taken place after executing that statement, then it's considered to be an *ambiguous* definition. As an example, let's consider the following Fortran fragment :

```
 1. subroutine A ( f1, f2 )
 2.         integer :: m, n
 3.         f2 = m * n
 4. end
 5. subroutine B ( f3, f4 )
 6.         call A ( f3, f3 )
 7. end
 8. program main
 9.         integer :: x
10.         call B ( x, 89 )
11.         read ( x )
12. end
```

The statements at lines 3 and 11 are unambiguous definitions since the value of expression m $\times$ n has been forced into f2, in line 3, while the value assigned to x in line 11 is

read from the keyboard. However, the calls at lines 6 and 10 are ambiguous definitions of f3 and x respectively. This is so because procedure A changes it formal parameter f2. The call at 6 causes f3 to be passed as the actual parameter for f2, thereby implying that f3 may be modified by this call. Since x has been passed as the actual parameter for formal f3 of procedure B, at line 10, we see that the corresponding statement will be an ambiguous definition of x. That both these two definitions are considered to be ambiguous follows from the fact that neither the assignment at line 3 nor the call at line 6 need be executed at all times, since, at times, control flow may cause either of these statements to be bypassed.

4.3 Uses and definitions due to non-call statements

To begin with, we shall consider what variables are used or defined in statements which do not involve calls to procedures[2]. We shall also discuss what types of uses or definitions arise in this case – ambiguous or unambiguous.

4.3.1 Definitions

For each variable x, that is mentioned as an *l-value*[3] in a non-call definition statement s in a procedure p, where x may be a single variable or an array variable (where the *l-value* mentioned in s is an array element), we decide the type of definition according to three choices :

1. If x is a local variable of p, then s is

   (a) an *unambiguous* definition of x, and

   (b) an *unambiguous* definition of all local variables of p which are aliased to x by COMMON/EQUIVALENCE association.

2. If x is a global variable, then s is

   (a) an *unambiguous* definition of x, and

[2]Of course, this includes function calls, but we have to consider assignment statements involving function calls twice – once for the assignment operation, as a non-call statement, and again for the procedure call part.

[3]See Section 3.3 for an explanation of this term.

(b) an *unambiguous* definition of all global variables aliased to x by COMMON/EQUIVALENCE association and in scope of p, and

(c) an *ambiguous* definition of all formals of p aliased by formal parameter association to x or to any of its aliases by COMMON/EQUIVALENCE association, not necessarily in scope of p.

3. If x is a formal parameter of p, then s is

(a) an *unambiguous* definition of x, and

(b) an *ambiguous* definition of all global variables aliased to x by formal parameter association and in scope of p, and

(c) an *ambiguous* definition of all formal parameters of p aliased to x by formal parameter association.

4.3.2 Uses

For each variable x that is a single variable or an array and is used in a non-call statement we perform, *mutatis mutandis*, the same set of actions as said in the previous section on definitions, with *use* replacing *definition*.

4.3.3 Implementation notes

For each variable in a scoping unit p, we maintain four bit vectors in the symbol table entry for this variable. These bit vectors and their uses are as follows :

- *useStatementVector* : set of all statements unambiguously using this variable,
- *maybeUseStatementVector* : set of all statements ambiguously using this variable,
- *defStatementVector* : set of all statements unambiguously defining this variable, and
- *maybeDefStatementVector* : set of all statements ambiguously defining this variable.

Each of these vectors has as many bits as the number of statements in the program. To set up a mapping from a set of ordinal numbers to the statements themselves, we number each node of the CFG and set up an array, *StatementMapArray*, to contain pointers to the CFG nodes corresponding to these numbers.

4.4 Uses and definitions due to procedure calls

4.4.1 Unambiguous uses

For each procedure call we shall consider the appearance of a variable x in some actual parameter to this call as an unambiguous use of that variable. In fact, for this variable we shall be performing the same set of actions as mentioned in the subsection on handling uses due to non-call statements in the previous section. This has been done, because the constant folding algorithm can use this information to fold this use of this variable only if this were considered to be an unambiguous definition of the same.

4.4.2 Interprocedural analysis

In the case of procedure calls one needs to know what variables shall be used/defined by other procedures called by this procedure either directly or indirectly, through a chain of nested calls. This information can be obtained through interprocedural data flow analysis ([ASU86, pages 653-660]). Of course, the use/definition information so obtained will be of the *ambiguous* type.

Interprocedural change information

For each procedure p we compute a set *def*[p] which contains all formal parameters and global variables of p, which are definitely defined in p. The aim of this analysis phase is to compute a set *change*[p], for each procedure p, that gives us the set of formal parameters of this procedure and global variables, not necessarily in its scope, that this procedure may modify whenever it is executed, taking into consideration all calls made by it, directly and indirectly, and without considering *any* aliasing effect. The algorithm used here is the same as the one given in [ASU86, page 658], with slight modifications with an eye to the implementation aspect. This algorithm has been given in figure 4.1.

Interprocedural access information

This is analogous to the change information and is used to compute use information at a call site. For each procedure p we define a set *use*[p] to be a set of all formals and globals of p that have definite uses in p. Similarly, we wish to construct a set *access*[p] that gives us all variables that may be used whenever p is executed, including all direct and indirect calls

```
1. For each procedure p do {
2.         change[p] = def[p]
3.         While changes occur to the change set of any of the procedures do {
4.                 For each procedure p do {
5.                         For each call site s in p do {
6.                                 Let q be the called procedure.
7.                                 Add all global variables in change[q] to change[p].
8.                                 For each data item x, which is either a formal parameter of
                                   p or a global variable, which has been passed as the actual
                                   parameter in location f of q at this call site, such that
                                   f ∈ change[q], do
9.                                         Add x to change[p].
                           }
                   }
           }
   }
```

Figure 4.1: Computing interprocedural change information.

made by it and ignoring all aliasing effects. The algorithm for this is a *mutatis mutandis* version of the previous one for computing interprocedural change information and so it has not been discussed here.

Implementation notes

In order to do interprocedural analysis we need to construct a *call graph*, having one node for each procedure in the program, with an edge from node p to node q for each call of the procedure corresponding to q by the procedure corresponding to p. As noted at the end of the previous chapter, we already have a list of call sites per procedure. Since each call site contains the name of the procedure being called, this list straightaway gives us the adjacency list in the call graph of the procedure containing this list. Moreover, the call sites themselves are contained in the entries of this list. Hence, the information in these lists are in a convenient form for interprocedural analysis.

We maintain four vectors in the global structure containing information about each procedure in the program. These vectors, along with their uses are :

- *changeVector* : *change* set,
- *defVector* : *def* set,
- *accessVector* : *access* set, and
- *useVector* : *use* set.

Each of these bit vectors have as many bits as the number of global variables and formal parameters in the entire program. The bit positions corresponding to the global variables are the same as their ordinal numbers. However, the bit positions corresponding to the formals are their ordinal numbers offset by the number of global variables in the program.

4.4.3 Using the interprocedural information

Using the change information

Consider each call statement s in a procedure p, with q being the called routine. We have two scenarios to consider in order to determine the set of variables modified by this statement. These are :

1. For each data item x passed as actual parameter at some location f of q, check if x is an *l-value*[4]. If it is so, and if f $\in$ *change*[q], then we decide the type of definition according to three choices :

   (a) If x is a local variable of p, then s is

      i. an *ambiguous* definition of x, and

      ii. an *ambiguous* definition of each local variable of p which is aliased to x by COMMON/EQUIVALENCE association.

   (b) If x is a global variable, then s is

      i. an *ambiguous* definition of x,

[4]See Section 3.3 for an explanation of this term.

ii. an *ambiguous* definition of each global variable, visible in p, which is aliased to x by COMMON/EQUIVALENCE association, and

iii. an *ambiguous* definition of each formal parameter of p aliased by formal parameter association either to x or to any global variable aliased to x be COMMON/EQUIVALENCE association.

(c) If x is a formal parameter of p, then s is

i. an *ambiguous* definition of x,

ii. an *ambiguous* definition of each global variable, visible in p, which is aliased to x by formal parameter association, and

iii. an *ambiguous* definition of each formal parameter of p, which is aliased to x by formal parameter association.

2. For each global variable g ∈ *change*[q] do

(a) If g is visible in p, then s is an *ambiguous* definition of q.

(b) For each global variable v which is visible in p and is aliased to g by COMMON/EQUIVALENCE association, s is an *ambiguous* definition of v.

Using the access information

This is, again, a *mutatis mutandis* version of the previous method, but used for computing the use information.

Implementation notes

The same four bit vectors, as mentioned in the section on non-call statements, shall be used here.

4.5 Reaching definitions

Again, we fall back to the standard algorithms for reaching definitions analysis, as given in [ASU86]. This method computes four vectors for each basic block in the CFG. These vectors and their significances are :

- **IN** : This set gives us the the statements whose definitions reach the *beginning* of this basic block. This set is computed by data flow analysis and is one among two outputs of the reaching definitions analysis.

- **OUT** : This is the set of statements whose definitions reach the *end* of the basic block under consideration. This set is the other output of the reaching definitions analysis.

- **GEN** : This is the set of statements which will definitely reach the end of this basic block, irrespective of whether they reach its beginning or not. This set can be computed using information on the nodes in this basic block alone.

- **KILL** : This gives us the set of statements whose definitions will definitely be prevented from reaching the end of this basic block. It may be noted that a definition which has been prevented from reaching a point is said to be *killed* at some place prior to that point, in data flow analysis parlance. This set too can be computed using the information on the nodes in this basic block alone.

The computation of these four sets follows entirely the method used in [ASU86]. There are, however, a couple of points of interest in this respect :

1. A single statement may unambiguously define a number of different variables. If this situation arises because a single variable is aliased to a number of others through COMMON/EQUIVALENCE association, then there is no problem, since they all represent the same locations anyway. However, if these variables are truly different variables, then the statement number, corresponding to this statement, no longer uniquely identifies the variable it modifies. Hence, one bit is not sufficient to represent this unambiguous definition [5].

   One solution to this problem is to have a different incarnation of this same statement for each variable defined by it, each with its own ordinal number and a corresponding bit, being assigned to it. However, this involves more programming effort. An easier way out is to transform all such statements into a series of statements, each defining only one of these variables. In Fortran 90, only a READ statement, with multiple

[5] Unless one assumes that this definition is not killed unless subsequent definitions of all variables, that it defines, require it to be killed. However, this can increase the number of definitions reaching a point, and so inhibit a number of optimizations.

number of parameters to be read into, can pose this problem. It is extremely easy to do this transformation on all READ statements in the program.

2. The above mentioned problem carries over to ambiguous definitions as well. Here, however, the number of variables being altered may be too much to allow the *one incarnation per variable modified* solution. Moreover, one cannot avoid this problem by any amount of transformation of the type mentioned previously. In fact, assigning one bit to this statement and killing it only if we have subsequent definitions of all variables that it may modify, each of them needing it to be killed, will lead to a barrage of reaching definitions and severely impede optimization. In order to solve this apparent *cul de sac* we shall resort to a further level of data flow analysis, which will refine the reaching definitions analysis already made. This is outlined in the following section.

4.5.1 Refinement of the reaching definitions information

The reaching definitions analysis shall be allowed to kill an ambiguous definition whenever any subsequent definition of even one of the variables, that this statement may define, require it to be killed. This can lead to an erroneous optimization as depicted in figure 4.2. Here statement 4 will kill 3 and 1, but 2 will pass through and reach 6. As we shall see later, the constant propagation algorithm will now erroneously fold this definition of y (if it happens to be a constant definition) to statement 6 despite the fact that y may have been redefined at 3 and this value may have been the value of y when control reaches 6.

This problem can be tackled if, whenever an unambiguous definition reaches a point, we have information as to whether it does so by passing through *at least one ambiguous definition* of the same variable or not (remember, ambiguous definitions *do not* kill other definitions). If this information were available, then at 6 we would know that 2 reaches 6 by passing through *some* ambiguous definition of y (the fact that it is actually statement number 3 is of no consequence). Hence, y will not be folded for, which is correct.

To gather the aforesaid information, we take the IN and OUT vectors computed in the reaching definitions analysis and do a refinement on it by a second data flow analysis. To do so, we define 3 sets, per basic block b, akin to the IN, OUT and GEN sets used in the reaching definitions analysis, as follows :

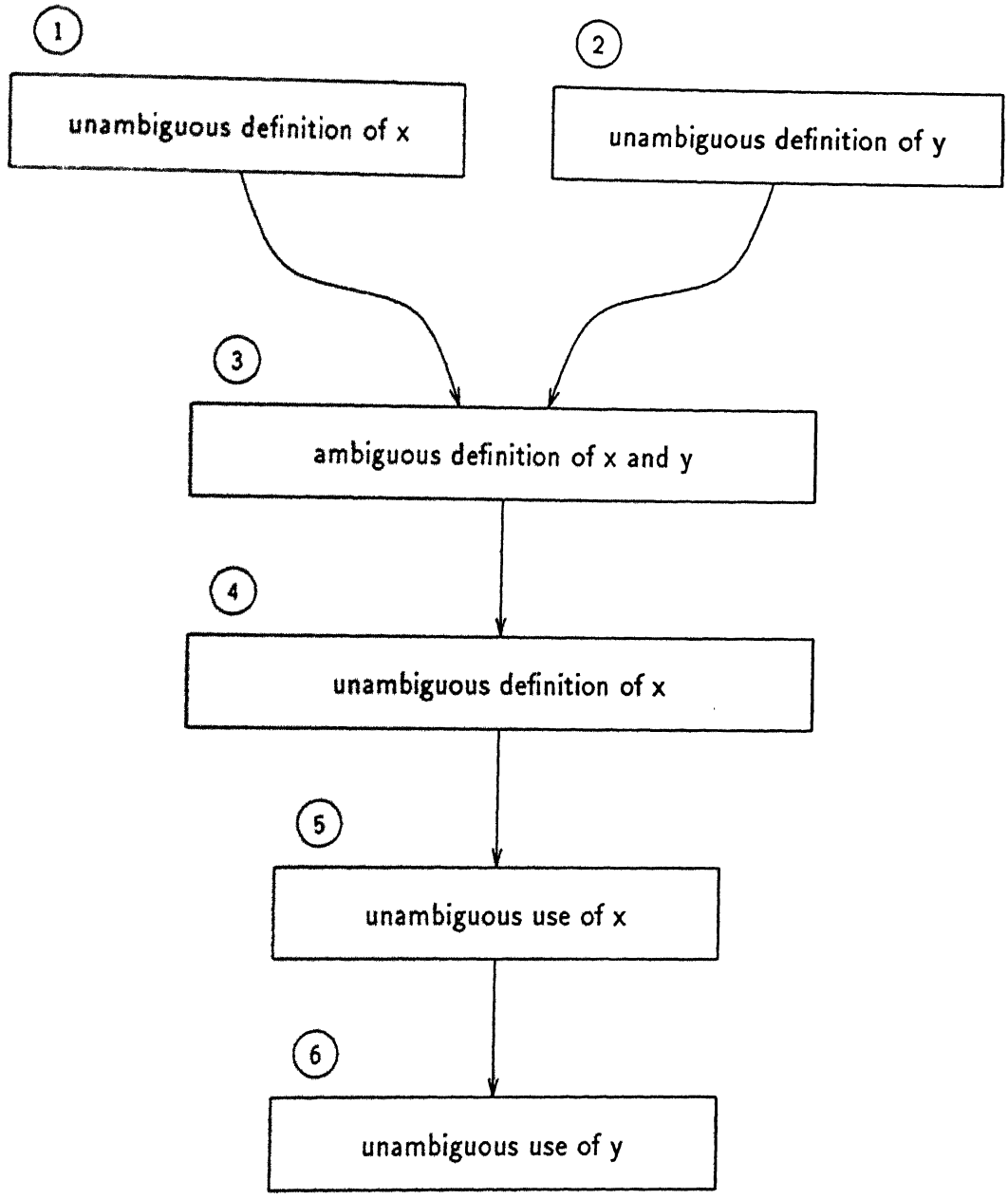


Figure 4.2: Illustrating error due to the killing of ambiguous definitions.

- **REFINED_IN** : This is the set of all unambiguous definitions reaching the beginning of b after passing through *at least one ambiguous definition* of the same variable.

- **REFINED_OUT** : This set is defined similarly to the previous one, but with respect to the end of the basic block.

- **REFINED_GEN** : This is a set of unambiguous definitions s, defining some variable x, such that one of the following is true :

  1. s is in b, *and*

     (a) s is the last *unambiguous* definition of x, or any of its COMMON/EQUIVALENCE aliases, in b, *and*

     (b) there is at least one *ambiguous* definition of x in b *strictly* after s.

  2. s is not in b, *and*

     (a) there is no *unambiguous* definition of x, or any of its COMMON/EQUIVALENCE aliases, in b, *and*

     (b) there is at least one *ambiguous* definition of x in b.

It must be noted that the KILL set does not need a refined counterpart. This is because the same set of definitions, as the ones in the KILL set used in reaching definitions analysis, constitute the set of definitions that will definitely get killed by the statements in a basic block even during the refinement phase. This is so because whether a definition of a variable is killed or not does not depend on whether it is followed by an ambiguous definition of the same variable or not.

By the preceding definition, we can see that REFINED_OUT(b) will contain a definition s of some variable x iff one of the following conditions is satisfied :

1. s is in REFINED_IN(b) and s is not in KILL(b)

   /* The same KILL set as used in reaching definitions analysis. */

2. s is in OUT(b) and s is in REFINED_GEN(b).

   /* OUT(b) is the result of the reaching definitions analysis. */

Condition 1 above states that s already satisfies the condition to be in REFINED_OUT(b), *viz.*, having at least one ambiguous definition of the same variable following it, without s itself getting killed in b.

Condition 2 above states that s reaches the end of the basic block, in terms of the reaching definitions analysis, and, over and above that, the required ambiguous definition following it has been introduced in this basic block itself.

Therefore, the data flow equations are :

$$\text{REFINED\_OUT(b)} = (\ \text{REFINED\_IN(b)} - \text{KILL(b)}\ ) \cup (\ \text{OUT(b)} \cap \text{REFINED\_GEN(b)}\ )$$

$$\text{REFINED\_IN(b)} = \bigcup_{p \in \textit{predecessor(b) in CFG}} \text{REFINED\_OUT(p)}$$

and the initialization for each basic block b being :

$$\text{REFINED\_OUT(b)} = \text{OUT(b)} \cap \text{REFINED\_GEN(b)}$$

4.5.2 Yet another problem

Consider the scenario shown in figure 4.3. Consider the case when only branch A, from the the starting node, is present, while branch B isn't. Statement 1 gets killed by 2 and the only definition of y reaching 4 is 3. Consequently, if 3 produces a constant definition of y, we shall *erroneously* fold for y. This is erroneous since 1 may define another value for y and then this definition should also have actually reached 4, but we are killing it here.

Moreover, if only branch B were present, and not A, then too the only definition of y leading to 4 would be 3. If this produces a constant, the resultant folding would be incorrect, since y would be undefined along the limb B.

To tackle both these problems, we make the start node of the CFG, of each procedure, an *unambiguous* definition of all variables in its scope. Then, in figure 4.3, 0 will also reach 4 and so prevent folding for y in 4, in both cases. However, this reintroduces the problem of multiple unambiguous definitions due to a single statement, as mentioned earlier. However, since we are doing this only for the start node of each CFG, we use the solution where we create one new incarnation of the start node, for each variable in this scoping unit, and allocate one bit for it in the statement vectors. The number of statements in each procedure shall now be increased by the number of variables (local variables, global variables and

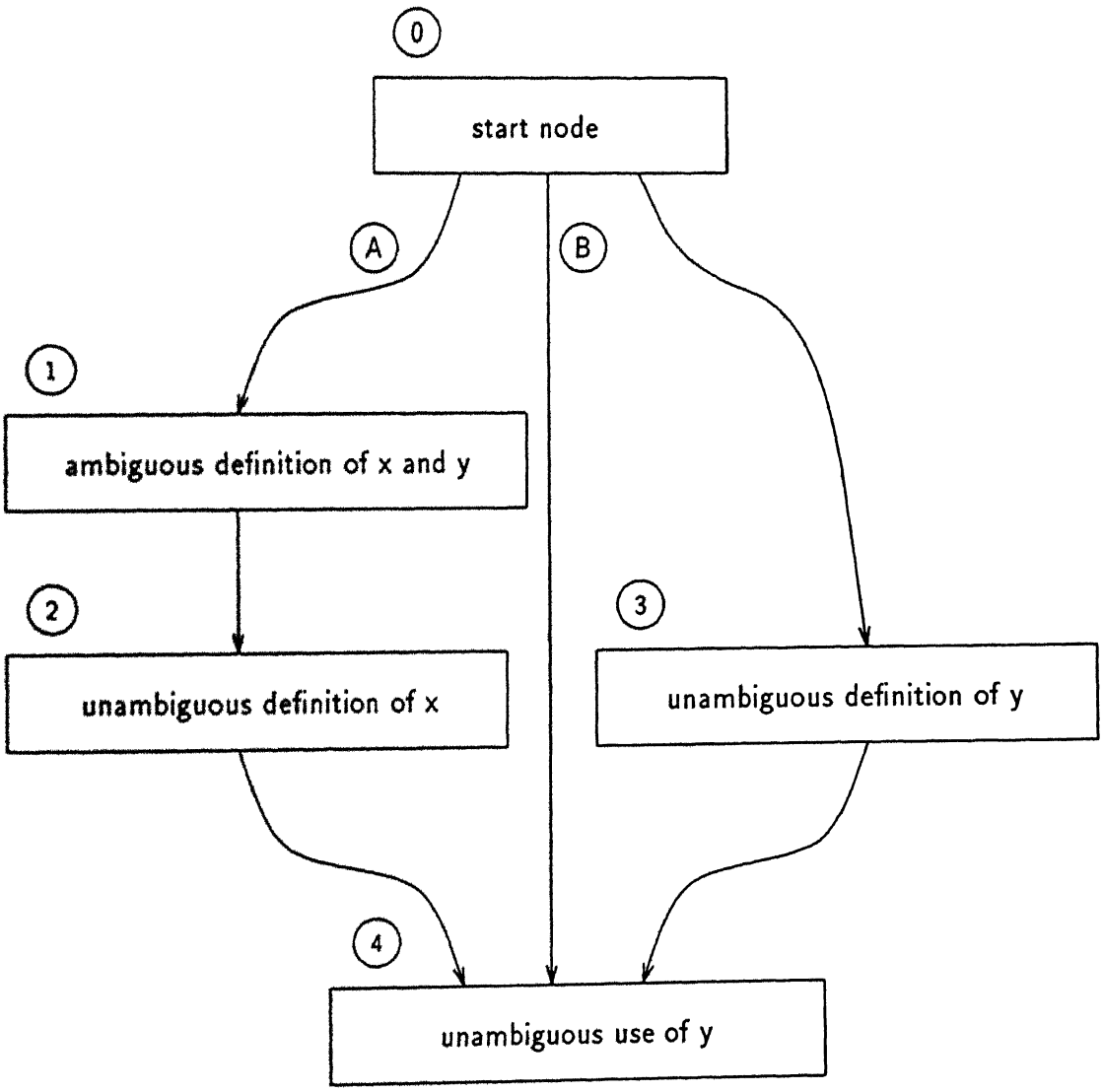


Figure 4.3: Illustrating error due to the absence of initialization.

formal parameters) in its scope. This is a small penalty to pay for avoiding erroneous optimizations. An important point to be noted is that these dummy initial assignments should not be used for constant propagation purposes.

4.5.3 Implementation notes

To facilitate the computation of the REFINED_GEN sets, we shall compute, for each statement which is an ambiguous definition of some variable, a bit vector with as many bits in it as the number of statements in the procedure (as augmented by the solution to the problem mentioned in the last section). This vector shall give us the set of *unambiguous* definitions that define the same variables as this ambiguous definition does. The set difference of this set and a set of unambiguous definitions known to be definitely killed till this statement, shall give us the contribution of this statement to the REFINED_GEN set for the basic block containing it. Of course, if this statement also happens to unambiguously modify the same variable as it ambiguously defines (for instance an assignment to some variable, which involves a function call which may modify the same variable) then it will itself get included in the REFINED_GEN set, which is erroneous. The algorithm to compute the REFINED_GEN set checks for this condition and explicitly resets the bit corresponding to this statements, if this is found to be the case.

Needless to say, all IN, OUT, GEN, KILL, REFINED_GEN, etc., shall use as many bits as the (augmented) number of statements in the procedure.

4.6 ud- and du-chains

Akin to du-chains, we can also construct a list of definitions of a variable reaching each use of the same. Such a list is called a *ud-chain*. In fact, the ud-chains are merely a different way of looking at the reaching definitions information. We do not explicitly construct the ud- and du-chains, as mentioned in [ASU86]. Instead, we compute these as and when they are needed. In fact, the use of bit vectors to represent the reaching definitions informations, combined with the use of an array to map ordinal numbers to pointers to the statements, forms an implicit construction of ud-chains that allows us to get to all definition statements reaching a particular use in one single operation.

However, to get to all uses reachable from a definition of a variable x, we must get to the symbol table entry for the variable x. Then we examine the bit vector(s) giving the uses of this variable. We go to each use of this variable and examine its reaching definitions vector. If the given definition is not in this vector, then we know that this use is not reached by this definition. Otherwise, this use is in the du-chain for this definition. We make the assumption that this is not too terribly inefficient since, ud-chaining is needed only for constant propagation and loop invariant computation and so we need to examine only the unambiguous uses of a variable. If this method is found to be too time consuming, then one can always use the above mentioned algorithm to construct the du-chains explicitly and store them, possibly, as bit vectors.

4.7 Constant propagation

The information gathered so far opens up avenues for a number of optimizations to be performed on the source code. The one that we have implemented is constant propagation and folding. This is particularly useful as an aid to the vectorizer, since this phase considers only DO-loops having constant initial, final and increment values for the loop variable, to be candidates for vectorization. Constant propagation may convert some of those DO-loops, hitherto considered to have variable values, into those having constant ones. This will enable us to discover more parallelism in the given program.

4.7.1 The algorithm

The algorithm for constant propagation used here is the simple algorithm described in Chapter 2, Section 2.3. This has been stated in figure 4.4 for each procedure p in the program. It must be noted that we shall be doing constant folding only for scalar variables, since the case of array elements is more involved and the returns from doing this do not justify the effort.

While folding the actual parameters in a procedure call, in step 9, one must exercise caution in two cases where a naive folding algorithm would lead to a change in the semantics of the program being optimized. These are :

1. When the actual parameter is a single variable and the corresponding formal parameter is in the *change* set of the called procedure, then one should not attempt to fold

```
1. Initialize constantDefSet to ∅.
2. For each unambiguous definition s in procedure p do {
3.      If s is a constant definition then
4.          Put s in constantDefSet.
   }
5. For each definition s in constantDefSet do {
6.      Let x be the variable being modified by s ( without considering any
        aliasing effects ).
7.      For each unambiguous use statment u of this definition of x do {
        /* Use du-chains to get to u. */
8.          If s is the only definition of x reaching u then {
            /* Use ud-chains to decide this. */
9.              Try folding all expressions in u containing textual references to
                x, by replacing x by its value defined in s, and evaluating
                constant expressions as far as possible.
10.             If u is an unambiguous definition of some variable and it now
                defines a constant value for it, then apply steps 6 through 10,
                using u in place of s.
            }
        }
   }
```

Figure 4.4: Algorithm for constant propagation.

this variable down to a constant value. This is so because this variable could be modified by this procedure call. If there's a subsequent use of this variable that uses this definition, then folding it down to a constant will remove this definition and therefore cause this statement to use the wrong value. Actually, one can still fold this single variable to a constant, provided that this definition does not reach any use at all (unambiguous or otherwise). However, this has not been done in this implementation.

2. When an actual parameter to a procedure call is an expression and this expression gets folded down to a single variable, and the corresponding formal parameter of the called procedure is in the *change* set of the called procedure, then we may again be

altering what the program is doing. If the called procedure were to change this formal parameter, then this variable would also get changed, which may not have been the case prior to folding this constant. Hence, what we have done is to explicitly check for such cases and convert such single variables into an expression (say, by adding 0 to it). This ensures that the proper value is passed to this formal parameter, while it forces the code generator to compute this expression and allocate a temporary memory location for it and pass its address to the formal parameter during this call (as is done for all actual parameters which are expressions).

The *constantDefSet* is, again, only a bit vector having as many bits as the number of statements in the procedure being considered.

Chapter 5

Integration, testing and epilogue

5.1 Integration

In the last two chapters we have seen how the aliasing analysis, reaching definitions computation and the constant propagation and folding algorithms have been implemented. In chapter 4 we had discussed about the building of the CFG from the syntax tree representation of the given program, which is the input to the optimization phase of the compiler. This automatically gives us the integration of this phase into the structure of the compiler (vide figure 1.1 of chapter 1 for this). In fact, in the various implementation notes we have also been mentioning the various data structures which are used in the implementation of this phase as also the construction of the information needed during this phase, such as the numbering of the variables and the formal parameters, the COMMON chains created during the parsing stage, *etc.* This phase expects its input to be in the form of a syntax tree of the input program, while its output is also the same. Such a design was undertaken because this phase is an optional one in the final compiler and hence, its input and output are expected to be of the same form.

5.2 Testing

The lack of standard benchmarking programs meant that all programs used in testing this phase had to be hand crafted. The compiler was run, with the optimization option on, on

programs meant to test a variety of scenarios in the algorithms used. Some of the more relevant ones are :

- Incorporation of variables in EQUIVALENCE to some variable in a COMMON block into this block.
- Setting of *low* and *high* information (explained in Section 3.4.2) for the root nodes of EQUIVALENCE trees.
- Construction of the formal parameter binding graph β and the formal pair binding graph π.
- Formation of the alias sets due to formal parameter binding.
- Integration of the results of COMMON/EQUIVALENCE alias analysis into the formal parameter aliasing algorithm.
- Uses and definitions due to non-call statements.
- Interprocedural analysis.
- Incorporating the alias information into the reaching definitions analysis.
- Refinement of the reaching definitions analysis.
- The following issues in constant propagation :
  - Folding an actual parameter to a single variable.
  - Folding an actual parameter from a single variable to a constant.
  - Folding arithmetic expressions.

Apart from the above, of course, the program had to be tested for integration with the rest of the system as well. However, this was the easiest part of the testing phase.

5.3 Summing up

In summary, the work outlined in this report concerns itself with the optimization phase of the Fortran 90 compiler, specifically the analysis of aliasing patterns in the program, the

computation of reaching definitions information and constant propagation and folding operations. This has been done at the source level by working with a syntax tree representation of the input Fortran 90 program. The level at which optimization has been done serves both as an aid to the vectorization process, that follows this phase, as well as a means towards generating efficiency in the program without disturbing the essential control structures in it.

5.4 Future work

The alias analysis phase is more or less complete, except in one or two places which has been mentioned in the program documentation. Nevertheless, these issues are very small and can be rectified without much effort. However, there are a number of areas where the current implementation has either not delved into or has adopted a less than efficient stand. Some of these are as follows :

- *Functions :*

  Functions have not been incorporated into the scheme of things, *i.e.*, as of now procedures in the input program comprise only of subroutines. This was done because Fortran needs disambiguation of function references and array references. Unfortunately, this has not been done during the type checking phase (in fact there's no explicit type checking phase in this compiler so far !). However, this disambiguation is not difficult and can be incorporated quite easily later on. The program has largely been written in order to aid future incorporation of functions into it.

- *Read transformation :*

  The transformation of read statements mentioned in Section 4.5 in the last chapter is yet to be done.

- *Incremental data flow analysis :*

  As mentioned in chapter 2, the efficiency of continued optimization depends on how efficiently one can maintain data flow information, such as reaching definitions, after each stage of the optimization process. In other words, it's necessary that some form of incremental data flow analysis be done in order to update such information. As of

now, this has not been done. However, some information, such as the interprocedural access information, has been computed as an aid for posterity.

- *Constant propagation :*

  A more efficient constant propagation algorithm, such as the one due to Wegman and Zadeck ([WeZa85]) should be used, in order to take into account the redundancy of certain conditional branches that can be evaluated at compile time.

- *Other optimizations :*

  The following phases of the optimizer have to be implemented in order to do total optimization on the input programs :

  - *Copy propagation.*
  - *Live variable analysis & dead code elimination.*
  - *Available expression computation and elimination of common subexpressions.*
  - *Loop optimizations such as loop invariant code motion, induction variable elimination and strength reduction.*

- *Back end :*

  The *back end* should be augmented in order to provide information to the user as to why certain optimizations (or, for that matter, vectorizations) have not been performed. This would bring out the true advantage of having such a feedback mechanism.

Bibliography

[AHU80] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, Reading, Mass.

[ASU86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers : Principles, techniques and tools.* Addison-Wesley, Reading, Mass.

[CCKT86] David Callahan, Keith Cooper, Ken Kennedy and Linda Torczon. *Interprocedural constant propagation.* Conf. Rec. Thirteenth ACM Symp. on Principles of Programming Langs., 1986, pp. 152-161.

[CFR89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and Frank K. Zadeck. *An efficient method of computing static single assignment form.* Conf. Rec. Sixteenth ACM Symp. on Principles of Programming Langs., 1989, pp. 25-35.

[CFR91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and Frank K. Zadeck. *Efficiently computing static single assignment form and the control dependence graph.* ACM Trans. on Programming Langs. and Systems, October 1991, 13(4),pp. 451-490.

[CoKe89] Keith D. Cooper and Ken Kennedy. *Fast interprocedural alias analysis.* Conf. Rec. Sixteenth ACM Symp. on Principles of Programming Langs., 1989, pp. 49-59.

[Coo85] Keith D. Cooper. *Analyzing aliases of reference formal parameters.* Conf. Rec. Twelfth ACM Symp. on Principles of Programming Langs., 1985, pp. 281-290.

[DRZ92] Dhananjay M. Dhamdhare, Barry K. Rosen and Frank K. Zadeck. *How to analyze large programs efficiently and informatively.* SIGPLAN '92, pp. 212-223.

[For8x] Fortran Forum. *Fortran 8x draft.* SIGPLAN Special Interest Publication on Fortran, Vol. 8, Number 4, December 1989.

[HS80] Ellis Horowitz and Sartraj Sahni. *Fundamentals of data structures.* Galgotia Booksource, New Delhi.

[LaRy92] William Landi and Barbara G. Ryder. *A safe approximate algorithm for interprocedural pointer aliasing.* SIGPLAN '92, pp. 29-41.

[MaRy90] Thomas J. Marlowe and Barbara G. Ryder. *An efficient hybrid algorithm for incremental data flow analysis.* Conf. Rec. Seventeenth ACM Symp. on Principles of Programming Langs., 1990, pp. 184-196.

[MoRe79] E. Morel and C. Renvoise. *Global optimization by suppression of partial redundancies.* Communications of the ACM, 22(2), pp. 96-103, February 1979.

[WeZa85] Mark N. Wegman and Frank K. Zadeck. *Constant propagation with conditional branches.* Conf. Rec. Twelfth ACM Symp. on Principles of Programming Langs., 1985, pp. 291-299.